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

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

January 28, 2015

Automatic Certificate Management Environment (ACME)
draft-barnes-acme-01

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 July 28, 2015.

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.

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

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

- The JWS MUST use the JSON or Flattened JSON Serialization
- If the JWS is in the JSON Serialization, it MUST NOT include more than one signature in the "signatures" array
- The JWS Header MUST include "alg" and "jwk" fields

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 a 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 recovery token that the client can provide later to associate a new account key pair in the event that the first account key pair is lost.

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 validate 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), a set of links to any required authorizations, 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.

```
Client                                              Server

CSR
Authorization URI(s)
Signature                    -------->

                             <--------          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.

```
Client                                              Server

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

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

Note that while ACME is defined with enough flexibility to handle different types of identifiers in principle, the primary use case addressed by this document is the case where domain names are used as identifiers. For example, all of the identifier validation challenges described in Section {identifier-validation-challenges} 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 certificate management functions that ACME enables:

- Registration
- 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.

5.1. Resources and Requests

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

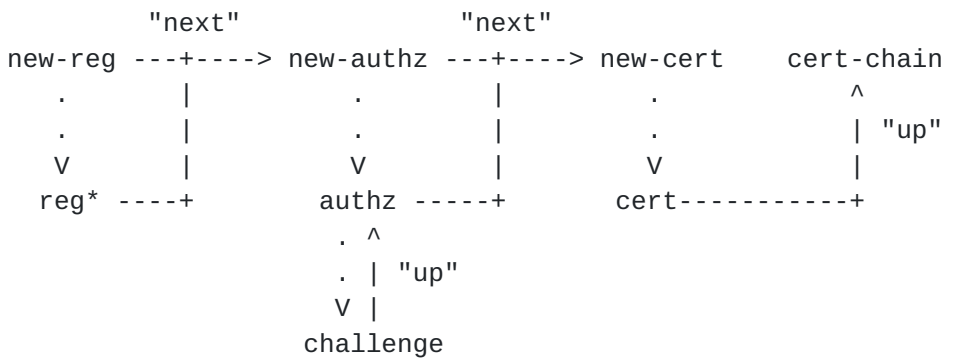
- Registration resources, representing information about an account key
- Authorization resources, representing an account key's authorization to act for an identifier
- Challenge resources, representing a challenge to prove control of an identifier
- Certificate resources, representing issued certificates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

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

In order to accomplish ACME transactions, a client needs to have the server's new-registration, new-authorization, and new-ceritificate URIs; the remaining URIs are provided to the client as a result of requests to these URIs. To simplify configuration, ACME uses the "next" link relation to indicate URI to contact for the next step in processing: From registration to authorization, and from authorization to certificate issuance. In this way, a client need only be configured with the registration 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 remainder of this section provides the details of how these resources are structured and how the ACME protocol makes use of them.

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.

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 issuane	POST new-cert	201 -> cert
Check for new cert	GET cert	200

5.2. 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 codes (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", "detail", and "instance" 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
malformed	The request message was malformed
unauthorized	The client lacks sufficient authorization
serverInternal	The server experienced an internal error
badCSR	The CSR is unacceptable (e.g., due to a short key)

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

5.3. Registration

An ACME registration resource represents a set of metadata associated to an account key pair, most importantly contact information and a recovery token. Registration resources have the following structure:

key (required, dictionary): The public key of the account key pair, encoded as a JSON Web Key object [I-D.ietf-jose-json-web-key].

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, or check with the client

on future authorizations (see the "recoveryContact" challenge type).

recoveryToken (optional, string): An opaque token that the client can present to demonstrate that it participated in a prior authorization transaction.

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 registration object containing only the "contact" field.

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

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

The server MUST ignore any values provided in the "key" or "recoveryToken" fields, 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 also provides a random recovery token. The server returns this registration object in a 201 (Created) response, with the registration URI in a Location header field. The server may also indicate its new-authorization URI using the "next" link relation.

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

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

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

  "recoveryToken": "uV2Aph7-sghuCcFVmvsiZw"
}
```

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" or "recoveryToken" 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, since these requests not authenticated.

5.4. Authorization Resources

An ACME authorization resource represents server's authorization for an account key pair to represent an identifier. In addition to a public key and 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 key is authorized to represent

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

  value (required, string): The identifier itself.

key (required, dictionary): The public key of the account key pair, encoded as a JSON Web Key object [I-D.ietf-jose-json-web-key].

status (optional, string): The status of this authorization. Possible values are: "pending", "valid", and "invalid". 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, dictionary): 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 key in the dictionary is a type of challenge, and the value is a dictionary with parameters required to validate the challenge, as specified in Section {identifier-validation-challenges}.

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.

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

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

  "key": { /* JWK */ },

  "challenges": [
    "simpleHttps": {
      "status": "valid",
      "validated": "2014-12-01T12:05Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
      "path": "Hf5GrX4Q7EBax9hc2jJnfw"
    },
    "recoveryToken": {
      "status": "valid",
      "validated": "2014-12-01T12:07Z",
      "token": "23029d88d9e123e"
    }
  ],
}
```

5.5. Key Authorization

The key authorization process establishes the authorization of an account key pair 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 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., 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 then the server issues challenges that the client responds to.

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 client MAY provide contact information in the "contact" field in this or any subsequent request.

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

{
  "identifier": {
    "type": "domain",
    "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.
- "key": the key used to verify the client's JWS request (i.e., the contents of the "jwk" field in the JWS header)
- "status": SHOULD be "pending" (MAY be omitted)
- "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": "domain",
    "value": "example.org"
  },

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

  "challenges": [
    {
      "type": "simpleHttps",
      "uri": "https://example.com/authz/asdf/0",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
    },
    {
      "type": "dns",
      "uri": "https://example.com/authz/asdf/1"
      "token": "DGyRejmCefe7v4NfDGDKfA"
    },
    {
      "type": "recoveryToken",
      "uri": "https://example.com/authz/asdf/2"
    }
  },

  "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 "simpleHttps" 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 "simpleHttps" challenge in the above authorization, it would send the following request:

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

{
  "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 response including the updated challenge.

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 key 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), remove any unfulfilled challenges, or add a "recoveryToken" field.

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. To provide some degree of control over polling, the server MAY provide a Retry-After header field to indicate how long it expect to take in finalizing the response.

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

HTTP/1.1 200 OK

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

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

  "key": { /* JWK */ },

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

  "challenges": [
    "simpleHttps": {
      "status": "valid",
      "validated": "2014-12-01T12:05Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
      "path": "Hf5GrX4Q7EBax9hc2jJnfw"
    },
    "recoveryToken": {
      "status": "valid",
      "validated": "2014-12-01T12:07Z",
      "token": "23029d88d9e123e"
    }
  ],
}
```

5.5.1. 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.

5.6. 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] and set of authorization URIs. The CSR encodes the parameters of the requested certificate; authority to issue is demonstrated by the JWS signature and the linked authorizations.

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.)

authorizations (required, array of string): An array of URIs for authorization resources.

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

{
  "csr": "5jNudRx6Ye4HzKEqT5...FS6aKdZeGsysoCo4H9P",
  "authorizations": [
    "https://example.com/acme/authz/asdf"
  ]
}
/* Signed as JWS */
```

~~~~~~~~~~

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 returns the certificate in a response with status code 201 (Created). The server MUST indicate a URL for this certificate in a Location header field.

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 can provide metadata about the certificate in HTTP headers. For example, the server can include a Link relation header field {{RFC5988}} with relation "up" to provide a certificate under which this certificate was issued. Or the server can include an Expires header as a hint to the client about when to re-query to refresh the certificate. (Of course, the real expiration of the certificate is controlled by the notAfter time in the certificate itself.)
~~~~~~~~~~

```
HTTP/1.1 201 Created Content-Type: application/pkix-cert Link:
https://example.com/acme/ca-cert ;rel="up";title="issuer" Location:
https://example.com/acme/cert/asdf

[DER-encoded certificate]
```

Certificate Refresh

The certificate URL (provided in the Location header of the server's response) is used to refresh or revoke the certificate. To refresh the certificate, the client simply sends a GET request to the certificate URL. This allows the server to provide the client with updated certificates with the same content and different validity intervals, for as long as all of the authorization objects underlying the certificate are valid.

If a client sends a refresh request and the server is not willing to refresh the certificate, the server MUST respond with status code 403 (Forbidden). If the client still wishes to obtain a certificate, it can re-initiate the authorization process for any expired authorizations related to the certificate.

Certificate Revocation

To request that a certificate be revoked, the client sends a POST request to the certificate URL. The body of the POST is a JWS object whose JSON payload contains an indication when the client would like the certificate to be revoked:

revoke (required, string):
: The time at which the certificate should be revoked. The value of this field MUST be either the literal string "now", or a date in RFC 3339 format {{RFC3339}}.

authorizations (required, array of string):
: An array of URIs for authorization resources.

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

{ "revoke": "now", "authorizations": [
"https://example.com/acme/authz/asdf" ] } /* Signed as JWS */
```

Before revoking a certificate, the server MUST verify that the account key pair used to sign the request 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 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" }
```

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.

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

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 str**
4. **Generate a self-signed certificate with a subjectAltName extension containing two dNSName va**
   1. **The domain name being validated**
   2. A name formed by hex-encoding Z and appending the suffix ".acme.invalid"
5. **Compute a nonce domain name by appending the suffix ".acme.invalid" to the nonce provided by**
6. **Configure the TLS server such that when a client presents the nonce domain name in the SNI f**

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 "**
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.

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.

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.

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.

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.

5.7. Other possibilities

For future work:

- Email
- DNSSEC
- WHOIS
~~~~~~~~~~

6. IANA Considerations

TODO

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

7. 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)

8. Acknowledgements

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.

9. References

9.1. 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), January 2015.

[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.

[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.

[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.

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

[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.

[RFC7386] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7386, October 2014.

9.2. Informative References

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

Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

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