# Technical Manual of LEEC: *Let's Encrypt Erlang with Ceylan*

**Organisation:** Copyright (C) 2020-2025 Olivier Boudeville

**Contact:** about (dash) leec (at) esperide (dot) com

**Creation date:** Wednesday, November 11, 2020

**Lastly updated:** Saturday, August 23, 2025

**Version:** 1.2.7

**Status:** Stable

**Dedication:** Users and maintainers of the `LEEC` library

**Abstract:** The role of the `LEEC` library is to interact from Erlang/OTP with servers implementing the ACME protocol - specifically *Let's Encrypt* servers, mostly in order to generate X.509 certificates.

The latest version of this documentation is to be found at the official LEEC website (`http://leec.esperide.org`).

The documentation is also mirrored here.

# Table of Contents

**Overview** 3

**Differences Introduced by this Fork** 3

**Prerequisites** 5
- Dependency Basics . . . 5
- Switching JSON Parsers . . . 5
- Dependency Issues between Webservers and HTTP(s) Clients . . . 5

**Building** 6

**Usage Example** 6
- Single-Domain Certificates with the `http-01` Challenge . . . 6
- Wildcard-Domain Certificates with the `dns-01` Challenge . . . 8
  - Limits of the Single-Domain Certificates . . . 8
  - Wildcard Certificates: Pros and Cons . . . 8
  - LEEC (Preliminary) Support of Wildcard Certificates . . . 9

**Design Notes** 14
- Multiple Domains Having Each Multiple Hostnames . . . 14
- Concurrent Certificate Operations . . . 14
- Let's Encrypt Accounts . . . 14

**Getting Information about the Generated Certificates** 14

**Other Files of Interest** 16

**Troubleshooting HTTPS Certificate-related Issues** 16

**Licence** 17

**Support** 17

**Possible Enhancements** 17

**Please React!** 17

**Ending Word** 17

## Overview

The LEEC library is the Ceylan fork of the original and much appreciated letsencrypt-erlang, which is a Let's Encrypt client library for Erlang, whose author is Guillaume Bour.

LEEC's purpose is to obtain proper X.509 security certificates from `Let's Encrypt` - or more generally from any ACME server (version 2) - to perform *Domain Validation*.

Such certificates are typically used in order to secure one's webserver so that it can offer a solid HTTPS connectivity, which is pretty much standard nowadays; yet they may also used for mail servers, FTP servers, and many more[1].

Three levels of certificates can be considered:

- **pure single-domain certificates**; for example a certificate applying only to `foobar.org`
- **single-domain certificates with SANs** (*Subject Alternative Name*); for example a certificate applying only to a domain (e.g. `foobar.org`) and related extra, predetermined DNS names (like `alpha.foobar.org`, `beta.foobar.org`, etc.)
- **wildcard certificates**; for example a certificate applying to a domain (e.g. `foobar.org`) and to *any* subdomain thereof (e.g. `*.foobar.org`)

Subdomains may be managed by different hosts or may correspond to virtual hosts, i.e. be managed by a single host that is able to multiplex the accesses that are made to them, based on the pseudo-host that the requester specified (then for example serving different web contents based on it).

Each certificate is obtained by proving to its ACME server of the issuer the ownership/control of a webserver or of a DNS entry.

At least currently, no wildcard certificate is managed as in-depth as it is done for per-domain certificates (more complex challenges would have to be implemented), yet it is nevertheless supported.

To ensure freshness, these certificates have intentionally a short lifespan (e.g. they last for 90 days); they shall therefore be renewed regularly and automatically from the certificate authority.

LEEC is notably used in the context of US-Web.

## Differences Introduced by this Fork

Compared to the original `letsencrypt-erlang` library, the main differences introduced by LEEC are:

- it is more specialised, in the sense that LEEC focuses on the "slave" use case (i.e. to be directly integrated within an Erlang webserver), as opposed to the "webroot" one (a third-party webserver is running separately with little possibilities of direct interactions) or the "standalone" one (where no specific prior webserver would be running, the certificate agent operating then its own one)

[1] For Email encryption and code signing, one may refer to this section regarding OpenPGP.

- more comments, more spell-checking, much clarification
- more typing, more runtime checking, extended traces supported
- security increased (notably using 4096-bit RSA keys)
- dependency onto Ceylan-Myriad added, to benefit from its facilities
- JSON parser can be JSX (the default), or Jiffy (refer to the Switching JSON Parsers section)
- HTTP client can be either `Shotgun` or the Erlang-native `httpc` client, to avoid any extra dependencies on `Gun` and `Cowlib` (whose versions could potentially clash with the ones required by any `Cowboy`-based integrating webserver)
- porting done from gen_fsm (soon to be deprecated) to the newer gen_statem
- minor API changes and additions, for a clearer and more flexible mode of operation
- fixed the compilation with Erlang version 24.0 and higher (e.g. w.r.t. to `http_uri`/`uri_string`, to updated dependencies such as Jiffy, and newer Cowboy for the examples)
- allow for *concurrent* certificate requests (e.g. if managing multiple domains with different keys, new certificates being requested for all of them at webserver start-up); so LEEC generates certificates in parallel and does not rely on a *registered* FSM (*Finite State Machine*) anymore
- global, ETS-based TCP connection pool replaced by an (optional) per-FSM internal cache (if relying on Shotgun)
- support for SAN (Subject Alternative Name) certificates, an extension to X.509 enabling a certificate to include a `subjectAltName` field to list, here, extra DNS names that are covered by this certificate
- basic support for the management of:
  - *Ephemeral Diffie-Helman* key, to ensure *Forward Secrecy* by relying on a set of keys that are never communicated
  - Intermediate Let's Encrypt Certificates
- SSL-based verification of the ACME peer, to avoid any man-in-the-middle attack (quite relevant when fetching security elements)
- first basic support of wildcard certificates

So, even if LEEC can be seen mostly as a "reckless" fork (in the sense that it became quickly obvious that retaining upstream compatibility could hardly be achieved) - with so many source-level differences (in terms of conventions, Myriad integration, whitespace cleanup) that a pull request can difficultly be considered - yet, in spite of the appearances, it remained quite close to the original (mainly differences of form) and followed the same structure.

By some ways, this LEEC fork is safer and more robust than the original, by others not (e.g. test coverage, autonomous use, continuous integration). A key goal was to make it easier to understand and maintain.

Most of the elements of this pull request from Marc Worrell have also been integrated.

# Prerequisites

## Dependency Basics

The general dependencies are:

- `openssl`, version 1.1.1 or higher (required to generate RSA key and certificate request)
- `Erlang/OTP` (refer to the Myriad prerequisite section section for the supported versions); in the context of LEEC, an Erlang version of at least `24.1` is strongly recommended in order to better handle alternate certificate chains

The LEEC-specific ones, which are automatically managed by rebar3 if opting for a rebar-based build, are:

- a JSON parser: either jsx (the default) or jiffy
- Ceylan-Myriad, for the various facilities on which LEEC relies
- optional: a more advanced HTTP client than the httpc Erlang-native one, namely Shotgun, which should be more efficient (TCP connection re-used, recent HTTP, etc.) at the cost of an extra dependency (which may clash with any your application may introduce, refer to the dependency issues section)

## Switching JSON Parsers

If wanting to switch from the default jsx to jiffy, following files shall be updated:

- rebar.config (knowing it is generated from conf/leec.app.src)
- src/leec.app.src (knowing it is a mere symlink to `ebin/leec.app`, which is itself generated from conf/leec.app.src)

(none in Myriad)

## Dependency Issues between Webservers and HTTP(s) Clients

A potential dependency problem is that many Erlang-based webservers are powered by Cowboy (thus Cowlib) whereas LEEC used to rely necessarily on Shotgun, thus on Gun (and thus Cowlib) as well. Most of the time this implied different (potentially incompatible) versions of Cowlib, whereas only up to one should exist in the code path at any time.

We prefer sticking to the Cowlib version that is induced by Cowboy. At the time of this writing, the latest Cowboy stable version (the one that webserver projects such as US-Web want) is 2.8.0 and relies on Cowlib 2.9.1, whereas the latest Shotgun stable version, 0.5.0, is lagging behind, relying on Gun 1.3.1, itself relying on Cowlib 2.6.0 (too old).

An attempt of solution was to remove the dependency of LEEC onto Shotgun (as it induced a dependency on an older Cowlib) but to use Gun instead, which is lower-level yet might be chosen in order to rely on the target Cowlib version. However we did not found a suitable Gun version for that (1.3 being too old, 2.0.* not ready).

So a last-resort solution has been to rely instead on the even lower-level Erlang-native httpc client module (involving `inets` and `ssl`). The result, although based only on HTTP/1.1 with no connection-reuse, proved satisfactory right from the start and thus is provided as an alternate way of using LEEC, without involving any extra dependency.

This allows embedding LEEC with only one dependency onto Myriad and one onto a JSON parser (either jsx or jiffy) - and no other one (top-level or induced).

# Building

Two build procedures can be used (from the root of LEEC), and are now mostly the same:.

- either a rebar3-based one; then run `make all-rebar3`, simply corresponding to:

  ```
  $ rebar3 upgrade --all
  $ rebar3 compile
  ```

- or one relying on Ceylan's native build system; once the relevant prerequisites have been setup (selected, downloaded, built), just run `make all`

This last procedure is the one that we prefer and use routinely (see the US-Web native deployment script as an example thereof).

# Usage Example

The main example of LEEC in action can be found in link with US-Web, whose sources can be found here; see notably class_USCertificateManager.erl and us_web_letsencrypt_handler.erl[2].

## Single-Domain Certificates with the `http-01` Challenge

This mode of operation, described in this section, is typical of the use case where an Erlang-based webserver (in this case based on Cowboy) has to automatically

[2]By the way, if you are currently reading LEEC's official website, then your browser already checked LEEC-obtained certificates!

renew certificates corresponding to the various virtual hosts (possibly dispatched under various domains) that it is making available.

A first part is to create as many LEEC FSMs as domains of interest, which will connect to the target ACME servers (most probably Let's Encrypt ones). Each FSM is a LEEC agent that will generate its own (strong) RSA key, create automatically its throwaway ACME account on the server, secure properly the communication (with TLS signatures, nonces, etc.), and wait for further user request regarding its domain of interest (e.g. `foobar.org`).

Such a request is bound to ask the ACME server to generate (as a Certificate Authority) a X.509 certificate covering, thanks to SAN, a set of subdomains (as FQDNs) to secure (e.g. `hello.foobar.org`, `hurricane.foobar.org`) - knowing that no wildcard certificate can be obtained with the `http-01` challenge being used here. The ACME server will send challenges to LEEC so that it can prove that it controls indeed each of these subdomains.

A second part of the LEEC action is to ensure that these answers are available indeed, as tokens. In practice the ACME server will attempt to read the information it communicated at the previous step from specific URLs (prefixed with `.well-known/acme-challenge/`) expected to be served for these subdomains (most probably thanks to virtual hosting). If the ACME server is able to query and read, directly from a domain, the right tokens corresponding to the challenges it sent for this domain, then the proof of actual control by the requester is established, and the ACME server can thus issue a corresponding certificate and transmit it appropriately to LEEC.

The overall webserver of the user shall thus track the transitions of these FSMs until (hopefully) they successfully complete their procedure and obtain from their ACME server the corresponding certificate. Then only the user webserver will be able to fire its https support with these brand new certificates[3].

Finally, a task scheduler may be used to trigger renewals on time (not too soon, not too late, as ACME rules apply and, of course, each FQDN shall be covered by a valid certificate at any time), and a task ring may be used to (paradoxically) ensure that the webserver as a whole does not interact too much in parallel (through its various LEEC FSMs) with the ACME server (despite hosting potentially a large number of FQDNs), knowing that severe rate limits (example in production) apply.

LEEC does its best to go through this procedure, validating as much as possible each of these steps for a better reliability/control, and reporting outcome for tracability and error management.

In practice, the user code is expected:

(A) to initialise first LEEC, with `leec:start/{1,2}` and proper options (see leec.erl); the PID of the corresponding LEEC FSM is then returned

(B) to request, thanks to this PID, a certificate to be generated for a domain, with `leec:obtain_certificate_for/{2,3}`

(C) to answer properly to the corresponding challenges for each (sub)domain, by delivering the right LEEC-computed tokens; see `leec:send_ongoing_challenges/2`

[3]Before, even if suitable certificates were pre-existing, at least the ACME URL prefix must remain over http instead of being automatically promoted to https as all others, otherwise no challenge could succeed.

(D) to poll this FSM to establish if/when the targeted certificate is available; actually it is more convenient to define in (2) a callback to be triggered by LEEC when appropriate

For US-Web, (1), (2) and (4) are managed by class_USCertificateManager.erl (see respectively `init_leec/5`, `request_certificate/1` and the `onCertificateRequestOutcome/2` callback). (3) is taken in charge by us_web_letsencrypt_handler.erl (see `init/2`).

## Wildcard-Domain Certificates with the `dns-01` Challenge

### Limits of the Single-Domain Certificates

As discussed in the previous section, obtaining single-domain certificates thanks to the `http-01` challenge is not straightforward, and has some drawbacks:

- as dedicated key pairs will be needed for each actual domain name (for `foo.bar.org`, `buzz.bar.org`, etc.), whenever having to host a fair number of websites (a few dozens?), you will have to generate as many of them; this will multiply the chances of experimenting failures and hitting the rate limits enforced by your ACME provider (typically Let's Encrypt); yet using SANs (*Subject Alt Names*) is a sure way to reduce considerably the number of certificates to be issued (only one for each of your "top-level" domains, regardless of the number of the virtual hosts they each federate)

- anyone curious enough will be able to access the SAN information listed in one's resulting SSL certificate (this is trivially done with most browsers) and collect all the DNS Names listed there, thus revealing them - whereas your DNS configuration may be set not to disclose them[4]; moreover, without this SAN information (needed in this scheme but then publicly available), we do not believe anyone could easily guess what are the domains that have been defined (and thus what are the websites that are hosted by a given "root" domain; this corresponds in our example to the `foo` and `buzz` domains of the `bar.org` root one[5])

### Wildcard Certificates: Pros and Cons

A more satisfactory solution is to rely on the more flexible wildcard certificates, that are certificates that can apply to *any* subdomain (e.g. `*.bar.org`) of a given root one. Fortunately, ACME providers like Let's Encrypt now offer them as well[6].

[4]Note that, regardless of the settings in terms of DNS and certificates, with SNI (i.e. *Server Name Indication*, the method to multiplex virtual hosts onto a single actual host), the specific visited virtual hostname (e.g. `foo`, in `foo.bar.org`) is not encrypted, and thus might be known of an eavesdropper.

[5]Note that there is no need to specify explicitly in a DNS zone one's subdomain CNAME (e.g. `"foo IN CNAME bar.org."`): a wildcard entry (`"* IN CNAME bar.org."`) can be preferred (more flexible, and not disclosing any subdomain). It is quite complementary to wildcard certificates, in the sense that, then, subdomains have never to be listed outside of one's server.

[6]Of course such certificates may be purchased instead, but they tend to be quite expensive. Moreover one may prefer to favour a free Internet, able to operate as much as possible without black-box, commercial solutions.

This implies far fewer keys and, ultimately, less information publicly disseminated; this implies also switching from a `http-01` challenge (where you prove that you control a webserver corresponding to a given domain) to a `dns-01` one (where you prove that you control the corresponding DNS zone entry).

This entails a few consequences:

- such a certificate generation mechanism has to update a DNS entry, which is probably more complex than updating a website accordingly
- this requires that the DNS provider for that domain supports such an automated update, through an API - bound to be specific to this provider
- modifying one's DNS entries automatically is fraught with security risks: the needed credentials shall not be accessed by third parties, otherwise they could complete a dns-01 challenge on their own to acquire new certificates for the associated domains and/or revoke your own existing certificates; yet various measures (credential file protection, IP-based restriction) can help reducing these risks

### LEEC (Preliminary) Support of Wildcard Certificates

At least currently, LEEC supports wildcard certificates only very superficially, through the third-party (yet considered reliable and trustable, and free software) solution that is used, directly or not, by the vast majority of server installations - namely certbot. Care has been taken to streamline and secure the operation as much as possible (for example with as little permissions given as possible, not to jeopardise one's server).

> **Note**
>
> A better approach would have been to instead recode all exchanges directly from Erlang, with direct REST requests, like already done for the single-domain certificates.
> Although fully possible (moreover many LEEC existing elements could be re-used for this new validation challenge), this ideal approach would require significant development time, dedication and efforts. Any contribution welcome!

So, for the time being, LEEC merely integrates the well-known procedures described by many sources (for instance, in French: [1], [2], [3]). This is done here on Arch Linux, and for the OVH DNS provider (to which of course we are not affiliated in any way).

Additionally to the intrinsic advantages of wildcard certificates, with this approach we moved:

- (probably) from a leaf certificate to a more flexible full-chain one
- from a 4096-bit RSA algorithm to a more recent, supposedly stronger, smaller 256-bit id-ecPublicKey (*ASN1 OID: prime256v1 / NIST CURVE: P-256*) one

One should nevertheless note that:

- rate limits still apply, and moreover certbot instances are not able to run concurrently (reporting "*Another instance of Certbot is already running*"), most probably because they are not able to edit collaboratively the certbot state on the filesystem; as a consequence, we use our task ring to avoid any overlapping, but then, on startup, the duration before all domains are ready may become large (as dns-01 challenges are long to resolve, because of DNS propagation; roughly 2 minutes and 15 seconds per domain, according to our tests)

- certbot manages by itself the certificates that it produces, reading their expiration date to decide at each launch whether they shall be then updated; as a result, when a LEEC-using program (like US-Web) starts, should it not check their remaining lifespan (which is currently not done), it shall better wipe out its local certbot persistent state (see `leec:reset_state/2`), so that it gets fresh certificates (otherwise these programs may plan a renewal too late for any certificate that was already generated)

**Software Installation** The certbot mechanism will have to use the API that is specific to one's DNS provider, in our case it will be certbot-dns-ovh.

Installation can be done with `pacman -S certbot certbot-dns-ovh`.

**DNS Configuration** The goal here is to be able to update automatically (by program) the relevant DNS settings of one's target domain (so that the ACME challenge can be written there).

For that, once a key has been created (through the web interface of the DNS provider - here, OVH), all operations (including key management) are to take place through REST calls that one can issue once for all, before having certbot be able to operate on its own.

**Creation of an API Key** For each domain, the credentials can then be obtained thanks to, here, the OVH createToken API function. The `Create API Keys` form will request:

- an application name (e.g. "LEEC for foobar.org")

- an application description (e.g. "LEEC YYYYMMDD update token for foobar.org")

- a validity duration (`Unlimited` if wanting to automate)

- the GET/POST/DELETE operations to enable for this token (replace below `YOUR_DOMAIN` by your actual domain, like `foobar.org`):

| **HTTP Request Type** | **Parameter** |
|---|---|
| GET | /domain/zone/ |
| GET | /domain/zone/YOUR_DOMAIN/ |
| GET | /domain/zone/YOUR_DOMAIN/status |

...continued on next page

| HTTP Request Type | Parameter |
|---|---|
| GET | /domain/zone/YOUR_DOMAIN/record |
| GET | /domain/zone/YOUR_DOMAIN/record/* |
| POST | /domain/zone/YOUR_DOMAIN/record |
| POST | /domain/zone/YOUR_DOMAIN/refresh |
| DELETE | /domain/zone/YOUR_DOMAIN/record/* |

**Note**

Due to the aforementioned DNS security risks, we strongly recommend that a set of allowed IP addresses is specified in order to constrain the origin of the API request calls. By design one has already a fixed public IP for one's server, isn't it?
However, should one's server be hacked, the attacker may then control both one of the allowed IPs and also any local credential file...

**Deletion of an API Key** Unused keys/tokens (especially those of unlimited validity) shall be deleted. This cannot be done here directly from a dedicated, user-friendly account-based web interface, so REST calls will have to be issued (from an allowed IP, typically the one of the server of interest).

For that the `ApplicationID` of the target token to delete shall then be obtained thanks to... another token, and a relevant one (otherwise: "This call has not been granted").

So the first step is to create a suitable (limited-time) token, by pointing one's browser to this URL to request such a token.

Once the form filled and the token generated, the returned credentials information may then be temporarily stored in one's environment; for example (of course these are bogus values):

```
# Never recording credentials in the shell history of this (UNIX) user:
$ unset HISTFILE

# Application Key:
$ AK="3a2b42bc0caeb7d4"

# Application Secret:
$ AS="fb7472abcd8f9980210571ca502d793c"

# Consumer Key (attached to one's account):
$ CK="5b1226df0d5ea12192ae2c9642cf7b60"
```

Then the actual deletion can be done, by fetching the target `applicationId` token (alternatively the API web interface can be used):

```
# As a one-liner, to ensure that timestamp is fresh enough (otherwise: "Query
# out of time"):
```

```
#
$ METHOD=GET; QUERY="https://eu.api.ovh.com/1.0/me/api/application"; \
 BODY=""; TIMESTAMP="$(date +%s)"; \
 SHA="$(echo -n ${AS}+${CK}+${METHOD}+${QUERY}+${BODY}+${TIMESTAMP} \
 | shasum | cut -d ' ' -f 1)"; SIGNATURE="\$1\$${SHA}"; \
 curl -X ${METHOD} -H "Content-type: application/json" \
 -H "X-Ovh-Application: ${AK}" \
 -H "X-Ovh-Consumer: ${CK}" -H "X-Ovh-Signature: ${SIGNATURE}" \
 -H "X-Ovh-Timestamp: ${TIMESTAMP}" ${QUERY}
[212174,212168]

# Presumably the oldest one:
$ TOKEN_TO_DELETE=212168

# Checking the target token (validity not returned by this call):
$ METHOD=GET; \
QUERY="https://eu.api.ovh.com/1.0/me/api/application/${TOKEN_TO_DELETE}"; \
BODY=""; TIMESTAMP="$(date +%s)"; \
SHA="$(echo -n ${AS}+${CK}+${METHOD}+${QUERY}+${BODY}+${TIMESTAMP} \
| shasum | cut -d ' ' -f 1)"; \
SIGNATURE="\$1\$${SHA}"; curl -X ${METHOD} -H "Content-type: application/json" \
-H "X-Ovh-Application: ${AK}" \
-H "X-Ovh-Consumer: ${CK}" -H "X-Ovh-Signature: ${SIGNATURE}" \
-H "X-Ovh-Timestamp: ${TIMESTAMP}" ${QUERY}
{"applicationId":212168,"name":"LEEC-for foobar.org",
 "description":"LEEC foobar.org updater","status":"active",
 "applicationKey":"3a2b42bc0caeb7d4"}

# Deletion:
$ METHOD=DELETE; \
QUERY="https://eu.api.ovh.com/1.0/me/api/application/${TOKEN_TO_DELETE}"; \
BODY=""; TIMESTAMP="$(date +%s)"; \
SHA="$(echo -n ${AS}+${CK}+${METHOD}+${QUERY}+${BODY}+${TIMESTAMP} | shasum \
| cut -d ' ' -f 1)"; SIGNATURE="\$1\$${SHA}"; \
curl -X ${METHOD} -H "Content-type: application/json" \
-H "X-Ovh-Application: ${AK}" -H "X-Ovh-Consumer: ${CK}" \
-H "X-Ovh-Signature: ${SIGNATURE}" -H "X-Ovh-Timestamp: ${TIMESTAMP}" ${QUERY}
null

# Checking the deletion:
$ METHOD=GET; QUERY="https://eu.api.ovh.com/1.0/me/api/application"; \
BODY=""; TIMESTAMP="$(date +%s)"; \
SHA="$(echo -n ${AS}+${CK}+${METHOD}+${QUERY}+${BODY}+${TIMESTAMP} | shasum \
| cut -d ' ' -f 1)"; SIGNATURE="\$1\$${SHA}"; \
curl -X ${METHOD} -H "Content-type: application/json" \
-H "X-Ovh-Application: ${AK}" -H "X-Ovh-Consumer: ${CK}" \
-H "X-Ovh-Signature: ${SIGNATURE}" -H "X-Ovh-Timestamp: ${TIMESTAMP}" ${QUERY}
[213174]
```

**Certbot Configuration** Now that, for a given domain, we have a suitable token and credentials, these latter ones will have to be made available to certbot.

We do not want this third-party code to run as root, we prefer running it as a LEEC-specific user: typically, for any usage like US-Web, the specific user running the webserver.

We therefore create in a directory of choice (for example `/etc/xdg/universal-server/`) a `leec-ovh-credentials-for-foobar.org.txt` file whose content (of course still based on bogus values) is:

```
dns_ovh_endpoint = ovh-eu
dns_ovh_application_key = 3a2b42bc0caeb7d4
dns_ovh_application_secret = fb7472abcd8f9980210571ca502d793c
dns_ovh_consumer_key = 5b1226df0d5ea12192ae2c9642cf7b60
```

and assign to it proper permissions, for example:

```
$ chmod 400 leec-ovh-credentials-for-foobar.org.txt
$ chown web-srv:us-srv leec-ovh-credentials-for-foobar.org.txt
$ chattr +i leec-ovh-credentials-for-foobar.org.txt
```

**Certbot Execution** Then an interactive test can be conducted, for example first as root, with:

```
$ certbot certonly --dns-ovh --dns-ovh-credentials \
/etc/xdg/universal-server/leec-ovh-credentials-for-foobar.org.txt \
-d foobar.org -d *.foobar.org
```

This should succeed in creating the `/etc/letsencrypt/live/foobar.org/fullchain.pem` (the certificate file used by most server software) and the `/etc/letsencrypt/live/foobar.org/privkey.pem` symlink to the actual credentials (the private key associated to this certificate).

A second more involved test could be as the final user (e.g. `web-srv`), from the target directory (e.g. `/opt/universal-server/us_web-data/certificates`):

```
$ sudo -u web-srv certbot certonly --config-dir . --work-dir . \
--logs-dir . --non-interactive --agree-tos --email you@foobar.org \
--dns-ovh --dns-ovh-credentials \
/etc/xdg/universal-server/leec-ovh-credentials-for-foobar.org.txt \
-d foobar.org -d *.foobar.org
```

The following actual (PEM-encoded) files of interest that are then obtained (in `/etc/letsencrypt/{live,archive}/foobar.org`) are:

- `cert.pem`: the public key certificate file
- `privkey.pem`: the private key certificate file
- `chain.pem`: the certificate chain file
- `fullchain.pem`: the concatenation of `cert.pem` and `chain.pem`

# Design Notes

## Multiple Domains Having Each Multiple Hostnames

At least the ACME servers from Let's Encrypt enforce various rate limits (both in production environment and in staging one) that are fairly low, which leads to preferring requesting certificates only on a per-domain basis (e.g. globally for `foobar.org`) rather than on a per-hostname host basis (e.g. one for `baz.foobar.org`, another one for `hurricane.foobar.org`, etc., these hosts being virtual ones or not), as such requests would quickly become too numerous to respect these rate thresholds.

A per-domain certificate should then include directly its various hostnames as *Subject Alternative Names* (SAN entries).

With the `http-01` challenge type, no wildcard for such SAN hosts (e.g. `*.foobar.org`) can be specified, so all the wanted ones have to be explicitly listed[7].

So for example, with LEEC, the certificate for `foobar.org` (that would be managed by a dedicated LEEC agent) should list following SAN entries: `baz.foobar.org`, `hurricane.foobar.org`, etc.

For more flexibility a wildcard certificate may be preferred.

## Concurrent Certificate Operations

LEEC implemented independent (`gen_statem`) FSMs to allow typically for concurrent certificate renewals to be triggered (thanks to autonomous LEEC agents, per-FSM connection pools, separate keys, etc.).

A drawback of the aforementioned Let's Encrypt rate limits is that, while a given FSM is to remain below said thresholds, a set of parallel ones may not.

Should this issue arise, an option is to use a single FSM and to trigger certificate requests in turn. Another one is to rely on a task ring in order to avoid by design that such FSMs overlap.

## Let's Encrypt Accounts

Currently LEEC creates automatically throwaway ACME accounts, which is convenient yet may prevent the use of CAA (*Certificate Authority Authorization*), if wanting to specify which Certificate Authorities (CAs) are allowed to issue certificates for one's domain.

# Getting Information about the Generated Certificates

If using LEEC to generate a certificate for a `baz.foobar.org` host, the following three files shall be obtained from the Let's Encrypt ACME server:

[7] As a result, the certificate may disclose virtual hosts that would be otherwise invisible from the Internet (as not even declared in the DNS entries for that domain that would act as wildcard name resolvers).

- `baz.foobar.org.csr`: the PEM certificate request, sent to the ACME server (~980 bytes)
- `baz.foobar.org.key`: the TLS private key regular file, generated on the server host, and kept there (~1675 bytes)
- `baz.foobar.org.crt`: the PEM certificate itself of interest (~3450 bytes), to be used by the webserver afterwards

To get information about this certificate:

```
$ openssl x509 -text -noout -in baz.foobar.org.crt

Certificate:
   Data:
       Version: 3 (0x2)
       Serial Number:
           04:34:17:fd:ee:9b:bd:6b:c2:02:b1:c0:84:62:ed:a6:88:5c
       Signature Algorithm: sha256WithRSAEncryption
       Issuer: C = US, O = Let's Encrypt, CN = R3
       Validity
           Not Before: Dec 27 08:21:38 2020 GMT
           Not After : Mar 27 08:21:38 2021 GMT
       Subject: CN = baz.foobar.org
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
               RSA Public-Key: (2048 bit)

              Modulus:
                   [...]
               Exponent: 65537 (0x10001)
       X509v3 extensions:
           X509v3 Key Usage: critical
               Digital Signature, Key Encipherment
           X509v3 Extended Key Usage:
               TLS Web Server Authentication, TLS Web Client Authentication
           X509v3 Basic Constraints: critical
               CA:FALSE
           X509v3 Subject Key Identifier:
               [...]
           X509v3 Authority Key Identifier:
               keyid:C0:CC:03:46:B9:58:20:CC:5C:72:70:F3:E1:2E:CB:20:B6:F5:68:3A

           Authority Information Access:
               OCSP - URI:http://ocsp.stg-int-x1.letsencrypt.org
               CA Issuers - URI:http://cert.stg-int-x1.letsencrypt.org/

           X509v3 Subject Alternative Name:
               DNS:hello.baz.foobar.org.crt, DNS:world.foobar.org.crt, DNS:somesite.fo
           X509v3 Certificate Policies:
               Policy: 2.23.140.1.2.1
```

```
                Policy: 1.3.6.1.4.1.44947.1.1.1
                  CPS: http://cps.letsencrypt.org

            CT Precertificate SCTs:
                Signed Certificate Timestamp:
                    Version   : v1 (0x0)
                    Log ID    : [...]
                    Timestamp : Jan  2 09:23:20.310 2021 GMT
                    Extensions: none
                    Signature : ecdsa-with-SHA256
                Signed Certificate Timestamp:
                    Version   : v1 (0x0)
                    Log ID    : [...]
                    Timestamp : Jan  2 09:23:20.320 2021 GMT
                    Extensions: none
                    Signature : ecdsa-with-SHA256
                                [...]
    Signature Algorithm: sha256WithRSAEncryption
    [...]
```

## Other Files of Interest

A `*.key` (e.g. `my-foobar-leec-agent-private.key`) file is a (PEM, strong enough) RSA private key generated by LEEC so that its agent can safely authenticate to the ACME servers it is interacting with.

`lets-encrypt-r3-cross-signed.pem` is the (PEM) certificate associated to the *Certificate Authority* (Let's Encrypt here). It is automatically downloaded by LEEC if not already available.

The `dh-params.pem` file contains the parameters generated by LEEC in order to allow for a safer *Ephemeral Diffie-Helman key exchange*, which is used to provide Forward Secrecy with TLS (see this article for further information). Its generation may take quite some time (but is to happen once).

## Troubleshooting HTTPS Certificate-related Issues

In order to understand why a given host (typically a webserver) does not seem to handle properly certificates, one may experiment with these commands from a client computer:

```
$ curl -vvv -I https://foobar.org
$ wget -v https://foobar.org -O -
$ openssl s_client -connect foobar.org:443
```

From the server itself:

```
$ iptables -nL
$ lsof -i:443
$ netstat -ltpn | grep ':443'
```

Third-party solutions might also be used, like testing your server with SSL Labs; thanks to LEEC, US-Web can be ranked "grade A" there.

## Licence

Ceylan-LEEC is distributed under the APACHE 2.0 licence, like the original work that it derives from.

## Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent through the project interface (typically issues), or directly at the email address mentioned at the beginning of this document.

## Possible Enhancements

- re-using ACME accounts: not creating throwaway, anonymous accounts but (possibly) reusing them by registering the ACME client with its email, etc.
- supporting certificate revocation
- supporting Elliptic Curve cryptography
- reintroducing elements brought by the upstream project yet not updated by the current fork: unit testing, standalone testing, hex package, various escripts and yml files involved
- besides the slave mode (main use case of interest with LEEC), better integrating/testing the other modes (webroot and standalone)
- supporting extra validation challenges, besides `http-01` and `dns-01` (necessary to obtain wildcard certificates, i.e. applying to all subdomains of a given domain without leaking their names through SANs), like `proof-of-possession-01`
- supporting directly other ACME services besides `Let's Encrypt` (like `ZeroSSL`)
- implementing for good `dns-01` rather than using certbot for that

## Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

One may also be interested in our mini-HOWTO regarding cybersecurity.

## Ending Word

Have fun with Ceylan-LEEC! (not supposed to involve any memory leak)

***LEEC***