cloudflare / cfssl

CFSSL: Cloudflare's PKI and TLS toolkit https://cfssl.org/

1,201 commits | 3 branches | 6 releases | 108 contributors | BSD-2-Clause

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

cbroglie Don't fail CI on development versions of Go — Latest commit ea4033a 3 days ago

api	Fix golint checks	4 months ago
auth	Inserted auth-key whitespace stripping after ReadFile	3 months ago
bundler	Fix remaining bundler tests	4 months ago
certdb	Revert accidental sqlite changes	4 months ago
certinfo	Format SKI properly in certinfo (#819)	a year ago
cli	Add basic stats interface	2 months ago
cmd	cfssljson: fixes parsing of bundle and root certificates (#842)	9 months ago
config	fix spelling mistake in config key content commitment	8 months ago
crl	CRL: Add a method for generating a CRL from the local DB	2 years ago
crypto	fix dead-url for crypto/pkcs7/pkcs7.go	2 years ago
csr	Add backdate param for initializing CA	10 months ago
doc	Update documentation for sums in newcert endpoint.	a year ago
errors	errors: add OCSPError case to errors.Wrap.	8 months ago
helpers	Add helpers for handling Ed25519 key (un)marshalling	3 months ago
info	Clean up some of the structures around info.	4 years ago
initca	fix spelling in comments	8 months ago
log	remove flag definition in init()	3 years ago
multiroot/config	fix unused value assignments caught by static checker (#718)	2 years ago
ocsp	Add basic stats interface	2 months ago
revoke	TUN-1126: Add a VerifyCertErr method which shows why verification failed	20 days ago
scan	Remove unnecessary refernce to internal/testenv	3 months ago
selfsign	Use stdlib CSR CheckSignature (#803)	a year ago
signer	Remove remnants of PKCS #11 support in README.md and comments	4 months ago
testdata	Add field to API docs and testdata.	3 years ago
transport	Remove unused backoff files	4 months ago
ubiquity	fix spelling in comments	8 months ago
vendor	Fix Travis CI builds	23 days ago
whitelist	Fix staticcheck failures.	2 years ago
.dockerignore	build to the dist folder	3 years ago
.gitignore	Fix documentation for how to build cfssl binaries	3 days ago
.travis.yml	Don't fail CI on development versions of Go	3 days ago
CHANGELOG	Adding a CHANGELOG.	3 years ago
Dockerfile	Checkout cfssl-trust in Dockerfile.	9 months ago
Dockerfile.build	Dockerfiles: update golang to 1.8.5	a year ago
Dockerfile.minimal	Update Dockerfile.minimal	9 months ago
Gopkg.lock	Fix Travis CI builds	23 days ago
Gopkg.toml	Fix Travis CI builds	23 days ago
LICENSE	Initial import.	4 years ago

Makefile	Fix documentation for how to build cfssl binaries	3 days ago
README.md	Fix documentation for how to build cfssl binaries	3 days ago
test.sh	Fix Travis CI builds	23 days ago

README.md

CFSSL

build passing codecov 56% godoc reference

CloudFlare's PKI/TLS toolkit

CFSSL is CloudFlare's PKI/TLS swiss army knife. It is both a command line tool and an HTTP API server for signing, verifying, and bundling TLS certificates. It requires Go 1.8+ to build.

Note that certain linux distributions have certain algorithms removed (RHEL-based distributions in particular), so the golang from the official repositories will not work. Users of these distributions should install go manually to install CFSSL.

CFSSL consists of:

- a set of packages useful for building custom TLS PKI tools
- the `cfssl` program, which is the canonical command line utility using the CFSSL packages.
- the `multirootca` program, which is a certificate authority server that can use multiple signing keys.
- the `mkbundle` program is used to build certificate pool bundles.
- the `cfssljson` program, which takes the JSON output from the `cfssl` and `multirootca` programs and writes certificates, keys, CSRs, and bundles to disk.

Building

Building cfssl requires a working Go 1.8+ installation and a properly set `GOPATH`.

```
$ git clone git@github.com:cloudflare/cfssl.git $GOPATH/src/github.com/cloudflare/cfssl
$ cd $GOPATH/src/github.com/cloudflare/cfssl
$ make
```

The resulting binaries will be in the bin folder:

```
$ tree bin
bin
├── cfssl
├── cfssl-bundle
├── cfssl-certinfo
├── cfssl-newkey
├── cfssl-scan
├── cfssljson
├── mkbundle
└── multirootca

0 directories, 8 files
```

Cross Compilation

You can set the `GOOS` and `GOARCH` environment variables to have Go cross compile for alternative platforms; however, cfssl requires cgo, and cgo requires a working compiler toolchain for the target platform.

Installation

Installation requires a working Go 1.8+ installation and a properly set `GOPATH`.

```
$ go get -u github.com/cloudflare/cfssl/cmd/cfssl
```

will download and build the CFSSL tool, installing it in `$GOPATH/bin/cfssl`.

To install any of the other utility programs that are in this repo (for instance `cfssljson` in this case):

```
$ go get -u github.com/cloudflare/cfssl/cmd/cfssljson
```

This will download and build the CFSSLJSON tool, installing it in `$GOPATH/bin/`.

And to simply install **all** of the programs in this repo:

```
$ go get -u github.com/cloudflare/cfssl/cmd/...
```

This will download, build, and install all of the utility programs (including `cfssl`, `cfssljson`, and `mkbundle` among others) into the `$GOPATH/bin/` directory.

Using the Command Line Tool

The `cfssl` command line tool takes a command to specify what operation it should carry out:

```
    sign             signs a certificate
    bundle           build a certificate bundle
    genkey           generate a private key and a certificate request
    gencert          generate a private key and a certificate
    serve            start the API server
    version          prints out the current version
    selfsign         generates a self-signed certificate
    print-defaults   print default configurations
```

Use `cfssl [command] -help` to find out more about a command. The `version` command takes no arguments.

Signing

```
cfssl sign [-ca cert] [-ca-key key] [-hostname comma,separated,hostnames] csr [subject]
```

The `csr` is the client's certificate request. The `-ca` and `-ca-key` flags are the CA's certificate and private key, respectively. By default, they are `ca.pem` and `ca_key.pem`. The `-hostname` is a comma separated hostname list that overrides the DNS names and IP address in the certificate SAN extension. For example, assuming the CA's private key is in `/etc/ssl/private/cfssl_key.pem` and the CA's certificate is in `/etc/ssl/certs/cfssl.pem`, to sign the `cloudflare.pem` certificate for cloudflare.com:

```
cfssl sign -ca     /etc/ssl/certs/cfssl.pem       \
           -ca-key /etc/ssl/private/cfssl_key.pem \
           -hostname cloudflare.com               \
           ./cloudflare.pem
```

It is also possible to specify CSR with the `-csr` flag. By doing so, flag values take precedence and will overwrite the argument.

The subject is an optional file that contains subject information that should be used in place of the information from the CSR. It should be a JSON file as follows:

```
{
    "CN": "example.com",
    "names": [
        {
            "C":  "US",
            "L":  "San Francisco",
            "O":  "Internet Widgets, Inc.",
            "OU": "WWW",
            "ST": "California"
        }
    ]
}
```

N.B. As of Go 1.7, self-signed certificates will not include the AKI.

Bundling

```
cfssl bundle [-ca-bundle bundle] [-int-bundle bundle] \
             [-metadata metadata_file] [-flavor bundle_flavor] \
             -cert certificate_file [-key key_file]
```

The bundles are used for the root and intermediate certificate pools. In addition, platform metadata is specified through `-metadata`. The bundle files, metadata file (and auxiliary files) can be found at:

```
https://github.com/cloudflare/cfssl_trust
```

Specify PEM-encoded client certificate and key through `-cert` and `-key` respectively. If key is specified, the bundle will be built and verified with the key. Otherwise the bundle will be built without a private key. Instead of file path, use `-` for reading certificate PEM from stdin. It is also acceptable that the certificate file should contain a (partial) certificate bundle.

Specify bundling flavor through `-flavor`. There are three flavors: `optimal` to generate a bundle of shortest chain and most advanced cryptographic algorithms, `ubiquitous` to generate a bundle of most widely acceptance across different browsers and OS platforms, and `force` to find an acceptable bundle which is identical to the content of the input certificate file.

Alternatively, the client certificate can be pulled directly from a domain. It is also possible to connect to the remote address through `-ip`.

```
cfssl bundle [-ca-bundle bundle] [-int-bundle bundle] \
             [-metadata metadata_file] [-flavor bundle_flavor] \
             -domain domain_name [-ip ip_address]
```

The bundle output form should follow the example:

```
{
    "bundle": "CERT_BUNDLE_IN_PEM",
    "crt": "LEAF_CERT_IN_PEM",
    "crl_support": true,
    "expires": "2015-12-31T23:59:59Z",
    "hostnames": ["example.com"],
    "issuer": "ISSUER CERT SUBJECT",
    "key": "KEY_IN_PEM",
    "key_size": 2048,
    "key_type": "2048-bit RSA",
    "ocsp": ["http://ocsp.example-ca.com"],
    "ocsp_support": true,
    "root": "ROOT_CA_CERT_IN_PEM",
    "signature": "SHA1WithRSA",
    "subject": "LEAF CERT SUBJECT",
    "status": {
        "rebundled": false,
        "expiring_SKIs": [],
        "untrusted_root_stores": [],
        "messages": [],
        "code": 0
    }
}
```

Generating certificate signing request and private key

```
cfssl genkey csr.json
```

To generate a private key and corresponding certificate request, specify the key request as a JSON file. This file should follow the form:

```
{
    "hosts": [
        "example.com",
        "www.example.com"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C":  "US",
            "L":  "San Francisco",
            "O":  "Internet Widgets, Inc.",
            "OU": "WWW",
            "ST": "California"
        }
    ]
}
```

Generating self-signed root CA certificate and private key

```
cfssl genkey -initca csr.json | cfssljson -bare ca
```

To generate a self-signed root CA certificate, specify the key request as a JSON file in the same format as in 'genkey'. Three PEM-encoded entities will appear in the output: the private key, the csr, and the self-signed certificate.

Generating a remote-issued certificate and private key.

```
cfssl gencert -remote=remote_server [-hostname=comma,separated,hostnames] csr.json
```

This calls `genkey` but has a remote CFSSL server sign and issue the certificate. You may use `-hostname` to override certificate SANs.

Generating a local-issued certificate and private key.

```
cfssl gencert -ca cert -ca-key key [-hostname=comma,separated,hostnames] csr.json
```

This generates and issues a certificate and private key from a local CA via a JSON request. You may use `-hostname` to override certificate SANs.

Updating an OCSP responses file with a newly issued certificate

```
cfssl ocspsign -ca cert -responder key -responder-key key -cert cert \
 | cfssljson -bare -stdout >> responses
```

This will generate an OCSP response for the `cert` and add it to the `responses` file. You can then pass `responses` to `ocspserve` to start an OCSP server.

Starting the API Server

CFSSL comes with an HTTP-based API server; the endpoints are documented in `doc/api/intro.txt`. The server is started with the `serve` command:

```
cfssl serve [-address address] [-ca cert] [-ca-bundle bundle] \
            [-ca-key key] [-int-bundle bundle] [-int-dir dir] [-port port] \
            [-metadata file] [-remote remote_host] [-config config] \
            [-responder cert] [-responder-key key] [-db-config db-config]
```

Address and port default to "127.0.0.1:8888". The `-ca` and `-ca-key` arguments should be the PEM-encoded certificate and private key to use for signing; by default, they are `ca.pem` and `ca_key.pem`. The `-ca-bundle` and `-int-bundle` should be the certificate bundles used for the root and intermediate certificate pools, respectively. These default to `ca-bundle.crt` and `int-bundle.crt` respectively. If the `-remote` option is specified, all signature operations will be forwarded to the remote CFSSL.

`-int-dir` specifies an intermediates directory. `-metadata` is a file for root certificate presence. The content of the file is a json dictionary (k,v) such that each key k is an SHA-1 digest of a root certificate while value v is a list of key store filenames. `-config` specifies a path to a configuration file. `-responder` and `-responder-key` are the certificate and the private key for the OCSP responder, respectively.

The amount of logging can be controlled with the `-loglevel` option. This comes *after* the serve command:

```
cfssl serve -loglevel 2
```

The levels are:

- 0 - DEBUG
- 1 - INFO (this is the default level)
- 2 - WARNING
- 3 - ERROR
- 4 - CRITICAL

The multirootca

The `cfssl` program can act as an online certificate authority, but it only uses a single key. If multiple signing keys are needed, the `multirootca` program can be used. It only provides the `sign`, `authsign` and `info` endpoints. The documentation contains instructions for configuring and running the CA.

The mkbundle Utility

`mkbundle` is used to build the root and intermediate bundles used in verifying certificates. It can be installed with

```
go get -u github.com/cloudflare/cfssl/cmd/mkbundle
```

It takes a collection of certificates, checks for CRL revocation (OCSP support is planned for the next release) and expired certificates, and bundles them into one file. It takes directories of certificates and certificate files (which may contain multiple certificates). For example, if the directory `intermediates` contains a number of intermediate certificates:

```
mkbundle -f int-bundle.crt intermediates
```

will check those certificates and combine valid certificates into a single `int-bundle.crt` file.

The `-f` flag specifies an output name; `-loglevel` specifies the verbosity of the logging (using the same loglevels as above), and `-nw` controls the number of revocation-checking workers.

The cfssljson Utility

Most of the output from `cfssl` is in JSON. The `cfssljson` utility can take this output and split it out into separate `key`, `certificate`, `CSR`, and `bundle` files as appropriate. The tool takes a single flag, `-f`, that specifies the input file, and an argument that specifies the base name for the files produced. If the input filename is `-` (which is the default), cfssljson reads from standard input. It maps keys in the JSON file to filenames in the following way:

- if **cert** or **certificate** is specified, **basename.pem** will be produced.
- if **key** or **private_key** is specified, **basename-key.pem** will be produced.
- if **csr** or **certificate_request** is specified, **basename.csr** will be produced.
- if **bundle** is specified, **basename-bundle.pem** will be produced.
- if **ocspResponse** is specified, **basename-response.der** will be produced.

Instead of saving to a file, you can pass `-stdout` to output the encoded contents to standard output.

Static Builds

By default, the web assets are accessed from disk, based on their relative locations. If you wish to distribute a single, statically-linked, `cfssl` binary, you'll want to embed these resources before building. This can by done with the go.rice tool.

```
pushd cli/serve && rice embed-go && popd
```

Then building with `go build` will use the embedded resources.

Additional Documentation

Additional documentation can be found in the "doc" directory:

- `api/intro.txt`: documents the API endpoints
- `bootstrap.txt`: a walkthrough from building the package to getting up and running