Passage de ce blog à Let's Encrypt

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 25 novembre 2018

https://www.bortzmeyer.org/passage-blog-lets-encrypt.html

Bon, comme la Terre entière, je suis passé à Let's Encrypt. Ce blog est désormais systématiquement en HTTPS pour tout le monde.

Il y a plus de quatre ans que ce blog est accessible en HTTPS <https://www.bortzmeyer.org/https-blog.html>, à la fois pour assurer la confidentialité (pas des réponses, puisque le contenu est public, mais des requêtes) et protéger contre toute modification en route. J'utilisais une autorité de certification gratuite, contrôlée par ses utilisateurs, et très simple à utiliser, CAcert <https://www.bortzmeyer.org/cacert.html>. J'en suis satisfait mais CAcert n'est pas intégré dans le magasin de certificats de la plupart des systèmes d'exploitation et/ou navigateurs Web. (Alors que des AC gouvernementales ayant déjà émis des faux certificats y sont, mais c'est une autre histoire.)

Cette non-présence dans les magasins d'AC obligeait les utilisateurs à ajouter CAcert manuellement, ce qu'évidemment peu faisaient. Résultat, je ne pouvais pas publier un URL en https:// sans recevoir des messages « c'est mal configuré », et je ne pouvais pas utiliser de bonnes pratiques comme de rediriger automatiquement les visiteurs vers la version sûre. D'où ce passage de CAcert à Let's Encrypt. La sécurité n'y gagne rien, mais ce sera plus pratique pour les utilisateurs. Notez que cela a des conséquences stratégiques pour l'Internet : la quasi-totalité des sites Web non-commerciaux (et beaucoup de commerciaux) utilisent la même AC, dont tout le monde est désormais dépendant.

Bon, il y a quand même un petit progrès technique, CAcert ne permettait pas de signer les certificats utilisant la cryptographie à courbes elliptiques (RFC 8422 [1]), alors que Let's Encrypt le permet. Voici ce certificat ECDSA avec la courbe elliptique P256, vu par le journal crt.sh <https://crt.sh/?id=944241000> (cf. RFC 6962), ou bien vu par GnuTLS :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8422.txt

```
% gnutls-cli www.bortzmeyer.org
- Certificate type: X.509
- Certificate[0] info:
 - subject 'CN=www.bortzmeyer.org', issuer 'CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US', serial 0x03
Public Key ID:
sha1:45600c1f3141cf85db95f5dac74ec1066bafb5b9
sha256:74d7df20684d3854233db36258d327dfce956720b836fd1f2c17f7e67ae84db9
Public key's random art:
+--[SECP256R1]----+
|      .+O+.... o.|
|       o.*..  * .|
|        . +o + +.|
|         .. o o+.|
|        S     .o=|
|              oo+|
|             . o.|
|                .|
|               E |
+-----------------+

- Certificate[1] info:
 - subject 'CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US', issuer 'CN=DST Root CA X3,O=Digital Signatu
- Status: The certificate is trusted.
```

Le certificat a été généré par :

```
% openssl ecparam -out blog.pem -name prime256v1 -genkey
% openssl req -new -key blog.pem  -nodes -days 1000 -subj '/CN=www.bortzmeyer.org' -reqexts SAN -config <(ca
```

La solution simple pour la deuxième commande (`openssl req -new -key blog.pem -nodes -days 1000 -out blog.csr`) n'était pas bonne car elle ne met pas de SAN ("*Subject Alternative Name*") dans le CSR, ce qui perturbe le client Let's Encrypt dehydrated <https://github.com/lukas2511/dehydrated/issues/598>.

La plupart des utilisateurs de Let's Encryt ne s'embêtent pas avec ces commandes OpenSSL. Ils utilisent un client Let's Encrypt comme certbot <https://certbot.eff.org/> qui fait ce qu'il faut pour générer la CSR et la faire signer par l'AC Let's Encrypt. Je ne l'ai pas fait car je voulais contrôler exactement le certificat. J'ai choisi le client Let's Encrypt dehydrated <https://github.com/lukas2511/dehydrated>. Après avoir vérifié la CSR :

```
% openssl req -text -in blog.csr
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: CN = www.bortzmeyer.org
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:34:ce:8a:50:e4:d0:bb:61:12:e6:39:98:cd:24:
                    13:59:47:83:bb:1c:5a:ae:96:be:49:d1:0f:cf:e0:
                    0b:96:b7:e6:fe:51:2c:ee:0f:bf:48:d4:73:5e:e5:
                    e5:79:0d:8e:f7:9b:5d:8d:d3:91:dd:fd:23:96:1f:
                    da:c2:46:03:b0
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        Attributes:
        Requested Extensions:
            X509v3 Subject Alternative Name:
                DNS:www.bortzmeyer.org
...
```

J'ai fait signer mon certificat ainsi :

```
% dehydrated --signcsr ./blog.csr > blog.crt
```

Une des raisons pour lequelles je voulais contrôler de près le certificat était que je veux publier la clé publique dans le DNS (technique DANE, RFC 6698), DANE étant une meilleure technique de sécurisation des certificats. J'ai donc un enregistrement TLSA :

```
% dig TLSA _443._tcp.www.bortzmeyer.org
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62999
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 7, ADDITIONAL: 17
...
;; ANSWER SECTION:
_443._tcp.www.bortzmeyer.org. 86400 IN TLSA 1 1 1 (
74D7DF20684D3854233DB36258D327DFCE956720B836
FD1F2C17F7E67AE84DB9 )
_443._tcp.www.bortzmeyer.org. 86400 IN RRSIG TLSA 8 5 86400 (
20181206024616 20181121161507 50583 bortzmeyer.org.
w04TM3ZKesaNrFrJMs9w4B8/V+vHDnaUxfO2lWlQTHZH
...
```

Et évidemment je l'ai testé avant de publier cet article :

```
% tlsa --verify www.bortzmeyer.org
SUCCESS (Usage 1 [PKIX-EE]): Certificate offered by the server matches the one mentioned in the TLSA record and
SUCCESS (Usage 1 [PKIX-EE]): Certificate offered by the server matches the one mentioned in the TLSA record and
SUCCESS (Usage 1 [PKIX-EE]): Certificate offered by the server matches the one mentioned in the TLSA record and
```

(On peut aussi tester en ligne <https://check.sidnlabs.nl/dane/>.) L'usage de DANE nécessite de ne pas changer la clé publique à chaque renouvellement du certificat <https://www.bortzmeyer.org/letsencrypt-certbot-keep-key.html> (ce que fait dehydrated par défaut). J'ai donc mis dans `/etc/dehydrated/config` :

```
PRIVATE_KEY_RENEW="no"
```

Let's Encrypt impose une durée de validité de trois mois pour le certificat. C'est court. Cela veut dire que le renouvellement doit être automatique. Par exemple, on met typiquement dans la configuration de cron un :

```
dehydrated --cron --hook /etc/dehydrated/hook.sh
```

Et tous les jours (dans mon cas), dehydrated va tourner, regarder les certificats dont l'expiration est dans moins de N jours (cf. paramètre `RENEW_DAYS` dans la configuration de dehydrated), les renouveller auprès de l'AC et exécuter les commandes situées dans `/etc/dehydrated/hook.sh`. De nombreux sites Web utilisant Let's Encrypt ont eu la mauvaise surprise de découvrir au bout de trois mois que leur certificat était expiré parce que le renouvellement n'avait pas marché (cron pas configuré, ou bien mal configuré). Le problème est d'autant plus fréquent que le discours « marketing » disant « Let's Encrypt, c'est super, tout est automatique » affaiblissait la vigilance des administrateurs système. Il est donc crucial, en plus de bien configurer son cron, de superviser l'expiration de ces certificats <https://www.bortzmeyer.org/tester-expiration-certifs.html>. Par exemple, ma configuration Icinga contient, entre autres :

```
vars.http_vhosts["blog-cert"] = {
    http_uri = "/"
    http_vhost = "www.bortzmeyer.org"
    http_ssl = true
    http_ssl_force_tlsv1_1_or_higher = true
    http_sni = true
    http_certificate = "7,4"
}
```

Avec cette règle, Icinga envoie un avertissement s'il reste moins de sept jours, et une alarme critique s'il reste moins de quatre jours de vie au certificat. De même, on doit superviser DANE <https://www.bortzmeyer.org/monitor-dane.html>.

Comme certains visiteurs du site Web essaient d'abord en HTTP (sans TLS), et comme les anciens liens en `http://` ne vont pas disparaitre du jour au lendemain, j'ai également mis en place une redirection, utilisant le code de retour HTTP 301 <https://http.cat/301> (RFC 7231, section 6.4.2). Dans la configuration d'Apache, cela donne :

```
<VirtualHost *:80>
ServerName www.bortzmeyer.org
        Redirect permanent / https://www.bortzmeyer.org/
</VirtualHost>
```

La redirection elle-même n'est pas sécurisée puisqu'on se connecte d'abord au serveur sans la protection qu'offre TLS. IL est donc prudent d'utiliser également HSTS (RFC 6797), pour dire « utilisez HTTPS systématiquement, dès le début ; je m'engage à ce qu'il reste actif ». Dans Apache, cela se fait avec :

```
Header set Strict-Transport-Security "max-age=7776000; includeSubDomains"
```

Comme tout le monde, j'ai testé la configuration TLS avec SSLlabs <https://www.ssllabs.com/>, CryptCheck <https://tls.imirhil.fr/> et Internet.nl <https://internet.nl/>. Un peu de "*gamification*" : SSLlabs me donne un A. CryptCheck me donne également un A (et me fait remarquer j'autorise le vieux TLS 1.0). SSLlabs, comme Internet.nl, me reprochent la durée trop courte de HSTS (c'est encore un peu expérimental).

À noter que j'ai conservé CAcert <https://www.bortzmeyer.org/cacert.html> pour les serveurs SMTP (pour lesquels on ne peut pas valider facilement avec Let's Encrypt, et, de toute façon, les

serveurs SMTP ont en général des certificats tellement problématiques que DANE - RFC 7672 - est la seule façon de les sécuriser). Même chose pour des sites internes, non accessibles depuis l'Internet et donc non vérifiables par l'AC Let's Encrypt.

Le remplacement automatique du certificat posait un autre problème : ce blog, `www.bortzmeyer.org` est sur deux serveurs (trois adresses IP en tout <`https://dns.bortzmeyer.org/www.bortzmeyer.org/ADDR`>). Il fallait donc recopier le nouveau certificat sur tous les serveurs. (Mais pas la clé privée qui, elle, est stable.) D'abord, j'ai choisi quel serveur ferait tourner dehydrated et donc recevrait le nouveau certificat. Ensuite, Let's Encrypt vérifie l'identité du serveur par un système de **défi** : lorsqu'il est sollicité (via le protocole ACME, normalisé dans le RFC 8555), Let's Encrypt génère un texte imprévisible qu'il envoie au client. Celui-ci doit alors le déposer à un endroit où Let's Encrypt pourra le récupérer. Donc, si une machine demande à l'AC Let's Encrypt un certificat pour `www.bortzmeyer.org`, Let's Encrypt va chercher si le texte est bien sur `https://www.bortzmeyer.org/.well-known/acme-challenge`. Le client prouve ainsi qu'il est légitime, qu'il peut recevoir les requêtes HTTP envoyées au nom. Le problème est que Let's Encrypt risque de se connecter sur un autre serveur que celui où le texte imprévisible a été déposé.

Pour éviter cela, j'ai configuré les autres serveurs pour relayer les requêtes HTTP allant vers `.well-known/acme-chal` en direction du serveur qui fait tourner dehydrated. Avec Apache, cela se fait avec le module mod_proxy <`https://httpd.apache.org/docs/2.4/fr/mod/mod_proxy.html`> :

```
ProxyRequests Off
ProxyPass /.well-known/acme-challenge/ https://ACME-MACHINE.bortzmeyer.org/.well-known/acme-challenge/
ProxyPreserveHost On
```

Cela permet à la machine qui lance dehydrated de toujours recevoir les requêtes de vérification, et donc de récupérer le certificat.

(Notez que la vérification par Let's Encrypt de l'identité du demandeur ne se fait pas forcément en HTTP. On peut aussi utiliser le DNS, ce qui serait une piste intéressante à explorer. Il y a aussi une méthode basée sur ALPN.)

Il reste ensuite à recopier le certificat nouvellement acquis sur tous les serveurs. J'ai utilisé SSH pour cela, avec du scp vers un compte sans mot de passe. Sur les serveurs Web, on crée le compte, et les répertoires où on copiera le certificat :

```
% sudo adduser copykey
% sudo -u copykey mkdir ~copykey/keys
% sudo -u copykey chmod 700 ~copykey/keys
```

Sur la machine ACME (celle où il y a dehydrated), on crée une clé SSH sans mot de passe (puisque tout doit pouvoir tourner depuis cron, pas d'interactivité) :

```
% ssh-keygen -t ed25519 -P ""
```

Et sur le serveur Web, on autorise les connexions depuis le détenteur de cette clé (root, a priori), mais seulement avec la commande scp vers le bon répertoire. On met dans `copykey/.ssh/authorized_keys` :

```
command="/usr/bin/scp -t keys",restrict ssh-ed25519 AAAAC3Nza... root@acme-machine
```

(Notez, et c'est amusant, que l'option `-t` indiquant le répertoire n'est apparemment pas documentée.)

Il reste alors, dans le fichier `/etc/dehydrated/hook.sh`, où se trouvent les commandes à exécuter lorsque le certificat est renouvellé, à effectuer la copie vers les serveurs Web :

```
function deploy_cert {
...
   for server in ${WEBSERVERS}; do
      scp /var/lib/dehydrated/certs/www.bortzmeyer.org/* ${REMOTEACCOUNT}@${server}:keys
   done
}
```

Enfin, dernière étape, les fichiers (notamment le certificat) ont été copiés sur le serveur Web mais il reste à les mettre là où le serveur les attend et à dire au serveur HTTP d'en tenir compte. On aurait pu copier les fichiers directement dans le répertoire final mais il aurait fallu être root pour cela et je n'avais pas envie de mettre `PermitRootLogin yes` dans le `sshd_config`, pour des raisons de sécurité. On copie donc vers un compte ordinaire (le `copykey` vu plus haut) et root sur le serveur Web a une tâche lancée par cron qui récupère les fichiers et relance le serveur HTTP :

```
% cat /etc/cron.d/copykeys
# Look for and copy TLS keys every 5 minutes
3-58/5 * * * *     root   /usr/local/sbin/copy-keys.sh

% cat /usr/local/sbin/copy-keys.sh
#!/bin/sh

OLDFILE=/etc/ssl/certs/www.bortzmeyer.org.fullchain.pem
NEWFILE=/home/copykey/keys/fullchain.pem

PATH=/sbin:/usr/sbin:${PATH}

OLDHASH=$(sha256sum ${OLDFILE} | cut -d' ' -f1)
if [ -z "${OLDHASH}" ]; then
    echo "Cannot find ${OLDFILE}" >&2
    exit 1
fi
NEWHASH=$(sha256sum ${NEWFILE} 2> /dev/null | cut -d' ' -f1 )
if [ ! -z "${NEWHASH}" ] && [ "${OLDHASH}" != "${NEWHASH}" ]; then
    cp -v ${NEWFILE} ${OLDFILE}
    apache2ctl graceful
fi
```