SGBLE

Geste de savoir

HAProxy et tout devient TLS !

3 mai 2021

Table des matières

Ce court billet montre comment se servir de HAProxy comme point de terminaison TLS devant n'importe quel serveur TCP, rendant par là même n'importe quel protocole «compatible» avec TLS. HAProxy est loin (vraiment très loin) de ne faire que ça, mais c'est le seul point abordé ici. Vous pourrez notamment apprendre comment transporter votre protocole fait maison sur une couche TLS facilement (et sans code).

Le billet est plutôt accessible. Ainsi seront clarifiées quelques notions sur la relation qu'entretiennent TCP, TLS et les protocoles applicatifs (HTTP, DNS, SMTP, …). Mais cela sans pour autant entrer dans les détails car il y aurait matière à rédiger un *big tuto* complet, autrement!

1. TLS, TCP, HTTPS et les autres

Quand on parle d'un protocole dans sa version sécurisée, par exemple HTTP**S**, on parle en réalité du protocole HTTP que l'on met sur une couche TLS[1]. HTTP**S** n'est en fait ni plus ni moins que ce que l'on aurait pu tout aussi bien appeler aussi *HTTP over TLS*. Cela vaut également pour les autres protocoles habituellement transportés sur TCP, y compris votre protocole maison qui pourrait passer sur TLS sans prévoir quoi que ce soit de particulier.

Pour nous en convaincre, utilisons `openssl` et faisons une petite expérience.
Vous vous êtes déjà peut-être amusés à communiquer en utilisant des protocoles «à la main» grâce à `telnet`, nous penserons notamment à HTTP ou SMTP. Faisons la même chose, mais cette fois en version sécurisée.

Allons donc sur https://zestedesavoir.com , sur le port 443/tcp (le port de HTTPS) grâce à la commande `openssl` qui va gérer la mise en place de la session TLS pour nous. Une fois la session établie, il nous laissera parler avec le serveur de façon habituelle comme nous l'aurions fait en HTTP simple. Regardons cela[2]:

```
% openssl s_client -brief -crlf -tls1_3 zestedesavoir.com:443
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN = zestedesavoir.com
```

```
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Server Temp Key: X25519, 253 bits
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx
Date: Fri, 30 Apr 2021 15:49:15 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 100424
Connection: close
Vary: Accept-Encoding
Vary: Cookie, Origin
Strict-Transport-Security: max-age=63072000; includeSubDomains;
    preload
X-Xss-Protection: 1
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
P3P: CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NAV"
X-Clacks-Overhead: GNU Terry Pratchett


<!DOCTYPE html>
<html class="enable-mobile-menu" lang="fr" antidoteapi_jsconne
...
...
...
```

(note: c'est à vous de taper le `GET / HTTP/1.0` pour obtenir la réponse HTTP qui s'ensuit. Le caractère représente un retour à la ligne.)

Nous voyons bien qu'une fois dans notre session TLS, nous communiquons en protocole HTTP classique comme nous le ferions avec `telnet`. Il n'y a rien de particulier, la partie HTTP ne sait même pas qu'il y a une couche une couche TLS sous ses pieds.[3]. Vous pouvez essayer la même chose sur un autre protocole facilement utilisable à la main, comme le SMTP par exemple.

TLS n'est en fait rien de plus que cela: un couche de sécurité sur laquelle on fait transiter normalement un protocole applicatif qui peut peut habituellement être transporté sur du TCP en clair. Il ne modifie en aucun cas le protocole applicatif pour le rendre sécurisé de façon magique car il s'agit simplement d'une couche intercalaire entre l'applicatif (HTTP) et le transport (TCP)[4].

Cela nous permet ainsi de comprendre ce qui suit.

2. Le problème

Dans mon précédent billet , je présentais la façon dont j'avais implémenté la résolution DNS en utilisant les résolveurs de Cloudlfare (`1.1.1.1`), la particularité étant l'utilisation de *DNS*

over TLS, aussi dit *DoT*.

Je déplorais le fait d'avoir recours à Cloudflare qui a certes le bon goût de proposer du DoT, mais qui reste malgré tout une dépendance externe hors de tout contrôle. J'ai donc remplacé Cloudflare par mon propre résolveur récursif complet qui fonctionne lui aussi avec DoT (d'ailleurs, il ne fonctionne *que* sous ce mode).

Le serveur DNS utilisé est l'excellent Unbound , logiciel libre de référence.
Les dernières versions d'Unbound supportent le mode TLS en serveur, proposant ainsi aux clients DNS le DoT. Hélas, la version dont je dispose n'est pas assez récente et ne prend donc pas cela en charge. Il faut donc ruser et c'est là que HAProxy entre en scène. (donnant ainsi une raison d'être au présent billet)

> Avant d'aller plus loin, précisons une chose importante. Contrairement à ce que l'on pense souvent, le protocole DNS ne fonctionne pas que sur UDP. Il fonctionne tout aussi bien sur TCP de façon parfaitement standard. Quand DNS fonctionne sur TCP, il utilise aussi le port 53 (53/tcp).

Cette précision est importante car le DoT fonctionne sur TCP.[5]. Vous avez compris, il s'agit en fait de simple DNS par dessus TLS, lui-même par dessus TCP . C'est exactement le même principe qu'avec le HTTP vu précédemment.[6]

Cependant, la nomenclature traditionnelle du suffixe *S* n'est pas en vigueur dans le cas du DoT: on ne parle pas d'un *DNSS* comme on parle d'un *HTTPS*, d'un *FTPS* ou encore d'un *SMTPS*. Reconnaissons ainsi à cette nouvelle nomenclature assez de mérite pour sa précision et son explicitation. En effet, un simple *S* pour *Secure*, voilà qui est finalement assez vague.

3. HAProxy comme terminaison TLS

HAProxy s'occupe de la *terminaison TLS* sur le port 853/tcp sur toutes les interfaces (dont publiques). Il transmet ensuite en clair les requêtes DNS (sur TCP) en clair à un *backend* (vocable de HAProxy) qui en l'occurrence sera Unbound, notre résolveur complet écoutant sur `127.0.0.1:53/tcp` ou `::1:53/tcp`. La réponse renvoyée par Unbound est ensuite remise dans sa couche TLS sur la même session que celle qui a été initiée par le client.

Voici ce que HAProxy nous permet de faire en image:

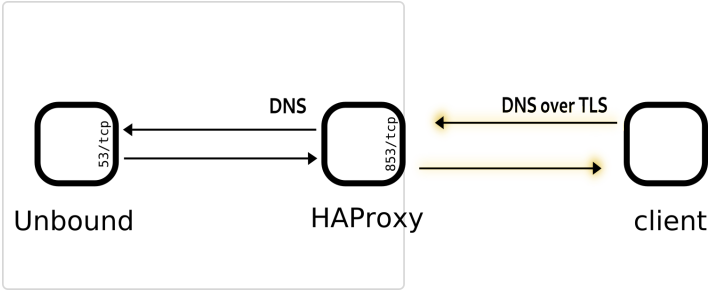

FIGURE 3.1. – La requête arrive en TLS sur le port 853/tcp, HAproxy transfère en clair le contenu au port 53/tcp local et renvoie la réponse en TLS au client

3. *HAProxy comme terminaison TLS*

Quand on veut mettre en place un service accessible sur TLS, les prérequis sont à peu près les mêmes que pour un service Web que l'on voudrait rendre accessible en HTTPS. Il faut se procurer un certificat TLS qui correspond à un ou des noms de domaine contrôlés. Ici pour l'exemple, considérons que ce nom sera `mon.dns.example.com.`. Bien entendu, Let's Ecnrypt est là pour ça si vous avez besoin d'un certificat. Personne n'a jamais dit que Let's Ecnrypt ne servait *que* pour le HTTPS 🍊

La configuration d'Unbound est assez simple. Elle n'écoute qu'en local, et l'on prend le soin de bien activer l'écoute sur le port 53/tcp qui sera utile à HAProxy.

```
server:
  # Écouter seulement sur les interfaces locales
  interface: 127.0.0.1@53
  interface: ::1@53

  do-tcp: yes  # important !
```

Du côté de HAProxy, est définit le *frontend* dénommé `dns-tls-in` dont nous parlions. Il écoute sur toutes les interfaces IPv4 et IPv6 sur port 853/tcp (c'est le `:::853 v4v6`) et mentionne le certificat qu'il utilisera pour communiquer avec TLS (`/path/to/certs/foobar.pem`). Le *backend* dénommé `dns-unbound` référence simplement le serveur Unbound que nous avons configuré. Il est indiqué que ce serveur sera accessible sur l'adresse locale `127.0.0.1`, port 53/tcp.

```
server:
  # Écouter seulement sur les interfaces locales
  interface: 127.0.0.1@53
  interface: ::1@53

  do-tcp: yes  # important !
```

Précisons un détail: HAProxy aime avoir le format de son certificat/clef en un seul fichier. Il s'agit d'une simple concaténation des fichiers habituellement générés par les outils tels que `certbot` ou `acme.sh`:

```
cat example.com.key fullchain.pem > foobar.pem
```

i Il est à noter que le résolveur DNS *backend* utilise le port 53/tcp. En réalité, n'importe quel port aurait pu être utilisé étant donné que le service n'est pas public. L'exemple ci-dessus marcherait parfaitement en utilisant le port 6969/tcp à la place, par exemple. Pour des raisons de cohérence, j'ai préféré garder le traditionnel port 53, m'assurant ainsi la compatibilité avec d'autres services locaux du serveur.

Le protocole DNS étant binaire, il nous sera hélas que trop pénible de l'écrire à la main dans une console `openssl` comme nous l'avons fait avec HTTP. Nous laisserons donc mon résolveur local de maison s'en charger pour nous. Voyons d'ailleurs sa configuration.

Elle ne change pas, par rapport au billet précédent. J'ai simplement changé les adresses de l'*upstream*, remplaçant celles de Cloudflare par celles de mon nouveau résolveur accessible via DoT.

```
forward-zone:
    name: "."
    forward-tls-upstream: yes
    forward-addr: "X.X.X.X@853#mon.dns.example.com"  # IPv4
    forward-addr: "X:X:X::X@853#mon.dns.example.com"  # IPv6

    # Au revoir, Cloudflare !
    #forward-addr: "2606:4700:4700::1111@853#cloudflare-dns.com"
    #forward-addr: "1.1.1.1@853#cloudflare-dns.com"
```

4. Et plein d'autres choses encore

Je suis content d'avoir eu ce souci pour présenter un cas d'usage de HAProxy en tant que terminaison TLS pour n'importe quel protocole sur TCP.

J'ai eu d'autres cas comme celui-là. La version de Redis 5 que j'utilisais encore jusqu'à récemment ne supporte pas le TLS. La version 6 a introduit le support. Il suffisait alors de mettre HAProxy devant et de faire en sorte que le client supporte lui aussi TLS (ce qui était déjà le cas du module `redis` de Python qui offre un mode `ssl=True`). J'ai donc pu établir des sessions TLS avec le protocole Redis entre client et serveur en restant pourtant sur la version 5 de Redis.

Bien entendu, il pourrait être préférable d'utiliser directement l'implémentation du serveur s'il le supporte: les versions d'Unbound récentes peuvent gérer le TLS, tout comme les serveurs Web classique tels que NGINX, Apache2. (je compte par ailleurs faire ainsi à terme)

Cependant, pour diverses raisons, la terminaison TLS découplée du serveur applicatif peut être intéressante à mettre en place et cela est une mission parfaite pour HAProxy.

Notons enfin que la terminaison TLS indépendante peut aussi présenter un intérêt de scalabilité. HAProxy est aussi très fort pour le *load-balancing* et *failover* entre plusieurs *backends*. Au lieu d'avoir le TLS activé sur chaque *backend*, seul HAProxy peut s'en charger et ensuite redistribuer le trafic derrière lui en clair. Cela a l'avantage de centraliser la gestion du certificat à un seul endroit et d'apporter plus de flexibilité et de simplicité sur les *backends* qui n'ont plus à s'en occuper. Que ce soit pour des services HTTP ou d'autres services TCP.

Pour rappel, HAProxy 2.0+ étend son support de TLS jusqu'à l'ALPN qui vous permet même de négocier le HTTP/2 directement si vous avez des services Web derrières. Un précédent billet en fait mention:

```
frontend main
    bind :::443 v4v6 ssl crt /path/to/cert.pem alpn h2,http/1.1
    mode http
    default_backend nginx

...
...
```

Ainsi s'achève ce billet.

5. Notes

1. Ou SSL, mais c'est du passé.
2. L'argument `-tls1_3` permet d'utiliser une version particulière de TLS, en l'occurrence la 1.3 qui est la dernière et la plus recommandée. Zeste de Savoir supporte TLS 1.3, ce qui en fait un bon élève!
3. C'est pour cette raison que quand vous avez besoin de savoir si le HTTPS a été utilisé dans votre application Web, vous devez généralement renseigner un *header* spécifique pour cela, typiquement `X-Forwarded-Proto`. Ce *header* est nécessaire justement parce que les couches sont agnostiques entre elles. HTTP ne sait pas—et n'a pas besoin de savoir—s'il repose sur une couche TLS ou s'il repose directement sur le transport TCP.
4. Si vous êtes familiers des modèles en couche (OSI, TCP/IP), nous pourrions dire que la couche TLS se situe entre la couche applicative et la couche de transport (en particulier TCP). Notons que cette conception peut néanmoins être mise en question par l'avènement du protocole QUIC qui est quelque peu spécial à cet égard du fait de son couplage entre le transport, la souche sécurité et presque jusqu'à l'applicatif. Mais il s'agit là d'une exception et ce modèle n'est pas (encore) la norme à ce jour. Le protocole QUIC est assez singulier pour mériter son propre article. J'aimerais vous en parler, un jour…
5. C'est une simplification. La réalité est quelque plus complexe et il existe aussi une version de TLS pour les protocoles de transport à base de datagrames comme UDP: DTLS. Par ailleurs, on peut techniquement faire du *DNS over DTLS* transporté sur UDP, le RFC existe bien , mais à ma connaissance cela n'est pas mis en œuvre. Dans son billet de blog à ce propos , Stéphane Bortzmeyer concluait ainsi en 2017, alors que l'idée était encore expérimentale (avant le RFC 8310):

> Il n'existe aucune mise en œuvre de DNS-sur-DTLS, et aucune n'est prévue. L'avenir de cette expérimentation est… incertain, à moins qu'un · e courageu · x · se développeu · r · se ne s'y mette?

Peut-être vous? 🍊
6. Une petite précision s'impose cependant: même si, comme nous l'avons vu, les couches se découplent assez bien, il se peut qu'un certain comportement de la couche sous-jacente soit explicitement attendue pour se conformer à un standard. C'est le cas du DNS over TLS tel que défini dans le RFC 7766 prônant l'utilisation d'une seule connexion TCP (et de surcroît une seule session TLS) si le client a plusieurs questions et réponses à recevoir.