Using SSL Certificates with HAProxy

Jul 29, 2014 (from https://serversforhackers.com/using-ssl-certificates-with-haproxy and https://www.haproxy.com/doc/aloha/7.0/deployment_guides/tls_layouts.html)

ssl http proxy

If you're interested in more of this type of content, check out the Servers for Hackers eBook!

Overview

If your application makes use of SSL certificates, then some decisions need to be made about how to use them with a load balancer.

A simple setup of **one** server usually sees a client's SSL connection being decrypted by the server receiving the request. Because a load balancer sits between a client and one or more servers, where the SSL connection is decrypted becomes a concern.

There are two main strategies.

SSL Termination is the practice of terminating/decrypting an SSL connection at the load balancer, and sending unencrypted connections to the backend servers.

This means the load balancer is responsible for decrypting an SSL connection - a slow and CPU intensive process relative to accepting non-SSL requests.

This is the opposite of **SSL Pass-Through**, which sends SSL connections directly to the proxied servers.

With SSL-Pass-Through, the SSL connection is terminated at each proxied server, distributing the CPU load across those servers. However, you lose the ability to add or edit HTTP headers, as the connection is simply routed through the load balancer to the proxied servers.

This means your application servers will lose the ability to get the `X-Forwarded-*` headers, which may include the client's IP address, port and scheme used.

Which strategy you choose is up to you and your application needs. SSL Termination is the most typical I've seen, but pass-thru is likely more secure.

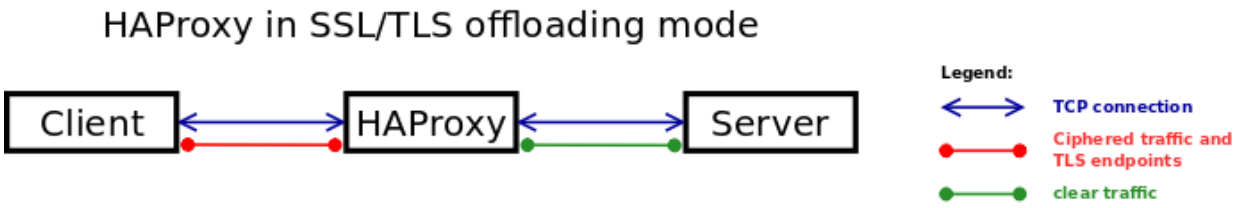
There is a combination of the two strategies **i.e SSL/TLS bridging or re-encryption**

, where SSL connections are terminated at the load balancer, adjusted as needed, and then proxied off to the backend servers as a *new* SSL connection. This may provide the best of both security and ability to send the client's information. The trade off is more CPU power being used all-around, and a little more complexity in configuration.

An older article of mine on the consequences and gotchas of using load balancers explains these issues (and more) as well.

HAProxy with SSL Termination / SSL/TLS offloading

We'll cover the most typical use case first - **SSL Termination**. As stated, we need to have the load balancer handle the SSL connection. This means having the SSL Certificate live on the load balancer server.



We saw how to create a self-signed certificate in a previous edition of SFH. We'll re-use that information for setting up a self-signed SSL certificate for HAProxy to use.

Keep in mind that for a production SSL Certificate (not a self-signed one), you won't need to generate or sign a certificate yourself - you'll just need to create a Certificate Signing Request (csr) and pass that to whomever you purchase a certificate from.

First, we'll create a self-signed certificate for `*.xip.io`, which is handy for demonstration purposes, and lets use one the same certificate when our server IP addresses might change while testing locally. For example, if our local server exists at 192.168.33.10, but then our Virtual Machine IP changes to 192.168.33.11, then we don't need to re-create the self-signed certificate.

I use the `xip.io` service as it allows us to use a hostname rather than directly accessing the servers via an IP address, all without having to edit my computers' Host file.

As this process is outlined in a passed edition on SSL certificates, I'll simple show the steps to generate a self-signed certificate here:

```
$ sudo mkdir /etc/ssl/xip.io
$ sudo openssl genrsa -out /etc/ssl/xip.io/xip.io.key 1024
$ sudo openssl req -new -key /etc/ssl/xip.io/xip.io.key \
                   -out /etc/ssl/xip.io/xip.io.csr
> Country Name (2 letter code) [AU]:US
> State or Province Name (full name) [Some-State]:Connecticut
> Locality Name (eg, city) []:New Haven
> Organization Name (eg, company) [Internet Widgits Pty Ltd]:SFH
> Organizational Unit Name (eg, section) []:
> Common Name (e.g. server FQDN or YOUR name) []:*.xip.io
> Email Address []:

> Please enter the following 'extra' attributes to be sent with your
certificate request
> A challenge password []:
```

```
> An optional company name []:
$ sudo openssl x509 -req -days 365 -in /etc/ssl/xip.io/xip.io.csr \
                    -signkey /etc/ssl/xip.io/xip.io.key \
                    -out /etc/ssl/xip.io/xip.io.crt
```

This leaves us with a `xip.io.csr`, `xip.io.key` and `xip.io.crt` file.

Next, after the certificates are created, we need to create a `pem` file. A `pem` file is essentially just the certificate, the key and optionally certificate authorities concatenated into one file. In our example, we'll simply concatenate the certificate and key files together (in that order) to create a `xip.io.pem` file. This is HAProxy's preferred way to read an SSL certificate.

```
$ sudo cat /etc/ssl/xip.io/xip.io.crt /etc/ssl/xip.io/xip.io.key \
           | sudo tee /etc/ssl/xip.io/xip.io.pem
```

When purchasing a real certificate, you won't necessarily get a concatenated "bundle" file. You may have to concatenate them yourself. However, many do provide a bundle file. If you do, it might not be a `pem` file, but instead be a `bundle`, `cert`, `cert`, `key` file or some similar name for the same concept. This Stack Overflow answer explains that nicely.

In any case, once we have a `pem` file for HAproxy to use, we can adjust our configuration just a bit to handle SSL connections.

We'll setup our application to accept both `http` and `https` connections. In the last edition on HAProxy, we had this frontend:

```
frontend localnodes
    bind *:80
    mode http
    default_backend nodes
```

To terminate an SSL connection in HAProxy, we can now add a binding to the standard SSL port 443, and let HAProxy know where the SSL certificates are:

```
frontend localhost
    bind *:80
    bind *:443 ssl crt /etc/ssl/xip.io/xip.io.pem
    mode http
    default_backend nodes
```

In the above example, we're using the backend "nodes". The backend, luckily, doesn't really need to be configured in any particular way. In the previous edition on HAProxy, we had the backend like so:

```
backend nodes
    mode http
    balance roundrobin
    option forwardfor
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    server web01 172.17.0.3:9000 check
```

```
    server web02 172.17.0.3:9001 check
    server web03 172.17.0.3:9002 check
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
```

Because the SSL connection is terminated at the Load Balancer, we're still sending regular HTTP requests to the backend servers. We don't need to change this configuration, as it works the same!

SSL Only

If you'd like the site to be SSL-only, you can add a `redirect` directive to the frontend configuration:

```
frontend localhost
    bind *:80
    bind *:443 ssl crt /etc/ssl/xip.io/xip.io.pem
    redirect scheme https if !{ ssl_fc }
    mode http
    default_backend nodes
```

Above, we added the `redirect` directive, which will redirect from "http" to "https" if the connection was not made with an SSL connection. More information on `ssl_fc` is available here.

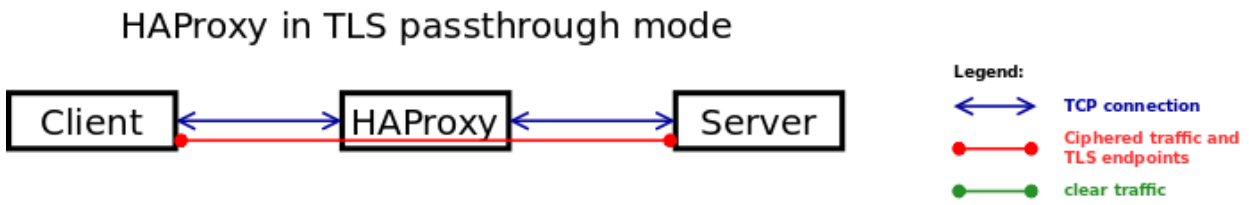
HAProxy with SSL Pass-Through / SSL/TLS passthrough

With **SSL Pass-Through**, we'll have our backend servers handle the SSL connection, rather than the load balancer.

The job of the load balancer then is simply to proxy a request off to its configured backend servers. Because the connection remains encrypted, HAProxy can't do anything with it other than redirect a request to another server.

In this setup, we need to use TCP mode over HTTP mode in both the frontend and backend configurations. HAProxy will treat the connection as just a stream of information to proxy to a server, rather than use its functions available for HTTP requests.

The picture below describes this layout:



First, we'll tweak the **frontend** configuration:

```
frontend localhost
    bind *:80
    bind *:443
    option tcplog
    mode tcp
    default_backend nodes
```

This still binds to both port 80 and port 443, giving the opportunity to use both regular and SSL connections.

As mentioned, to pass a secure connection off to a backend server without encrypting it, we need to use TCP mode (`mode tcp`) instead. This also means we need to set the logging to tcp instead of the default http (`option tcplog`). Read more on log formats here to see the difference between `tcplog` and `httplog`.

Next, we need to tweak our **backend** configuration. Notably, we once again need to change this to TCP mode, and we remove some directives to reflect the loss of ability to edit/add HTTP headers:

```
backend nodes
    mode tcp
    balance roundrobin
    option ssl-hello-chk
    server web01 172.17.0.3:443 check
    server web02 172.17.0.4:443 check
```

As you can see, this is set to `mode tcp` - Both frontend and backend configurations need to be set to this mode.

We also remove `option forwardfor` and the `http-request` options - these can't be used in TCP mode, and we couldn't inject headers into a request that's encrypted anyway.

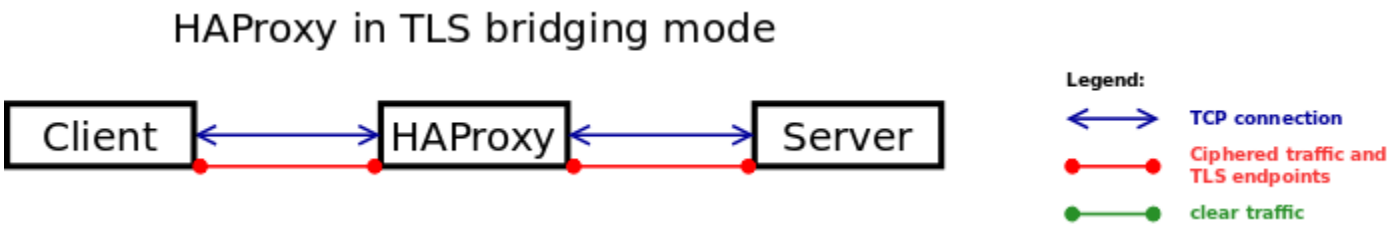
For health checks, we can use `ssl-hello-chk` which checks the connection as well as its ability to handle SSL (SSLv3 specifically) connections.

In this example, I have two fictitious server backend that accept SSL certificates. If you've read the edition SSL certificates, you can see how to integrate them with Apache or Nginx in order to create a web server backend, which handles SSL traffic. With SSL Pass-Through, no SSL certificates need to be created or used within HAproxy. The backend servers can handle SSL connections just as they would if there was only one server used in the stack without a load balancer.

SSL/TLS bridging or re-encryption

In this mode, **HAProxy** decipher the traffic on the client side and re-encrypt it on the server side. It can access to the content of the request and the response and perform advanced processing over the traffic.

The picture below describes this layout:



In this mode, **HAProxy** can run either in `mode tcp` or `mode http` and the keywords `ssl` and `crt` must be setup on the frontend's bind line and at least `ssl` on the backend's server line (crt is available, but optional).

The sample fetch methods which apply to this mode are the ones whose name starts by `ssl_c`, `ssl_f` `ssl_fc` and `ssl_bc`.

Examples:

1. Simple TLS bridging for an HTTPs application in order to perform cookie persistence:

```
frontend ft_myapp
 bind 10.0.0.1:443 ssl crt myapp
 mode http
 [...]
 default_backend bk_myapp

backend bk_myapp
 mode http
 [...]
 cookie MYAPP insert indirect nocache
 server app1 10.0.0.11:443 ssl check cookie app1
 server app2 10.0.0.12:443 ssl check cookie app2
```

2. Use **HAProxy** to export a weak SSLv3 service on internet over strong TLS1.2 protocol:

```
frontend ft_public
 bind 10.0.0.1:443 ssl crt myapp force-tlsv12
 mode tcp
 [...]
 default_backend bk_internal

backend bk_internal
 mode tcp
 [...]
 server sslv3server 10.0.0.11:443 check
```

Resources

- HAProxy Official blog post on SSL Termination
- SO Question: "What is a PEM file?"
- Reading custom headers in Nginx - Not mentioned in this edition specifically, but useful in context of reading `X-Forwarded-*` headers sent to Nginx
- So You Got Yourself a Load Balancer, an article about considerations to make in your applications when using a load balancer.