Tutorial 4 - Configuring an SSL server

What are we doing?

We are setting up an Apache web server secured by a server certificate

Why are we doing this?

The HTTP protocol uses plain text, which can very easily be spied on. The HTTPS extension surrounds HTTP traffic in a protective SSL/TLS layer, preventing snooping and ensuring that we are really talking to the server we entered in the URL. All data is sent encrypted. This still doesn't mean that the web server is secure, but it is the basis for secure HTTP traffic.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).
- Understanding of the minimal configuration in Tutorial 2 (Configuring a minimal Apache server).

Step 1: Configuring a server using SSL/TLS, but without an officially signed certificate

When contacted by a client, an SSL server must use a signed certificate to identify itself. For a successful connection the client must be familiar with the signing authority, which it does by checking the certificate chain from the server to the root certificate of the signing authority, the certificate authority. Officially signed certificates are acquired from a public (or private) provider whose root certificate is one the browser is familiar with.

The configuration of an SSL server therefore comprises two steps: Obtaining an officially signed certificate and configuring the server. The configuration of the server is the more interesting and easier part, which why we'll do that first. In doing so, we'll be using an unofficial certificate present on our system (at least if it's from the Debian family and the *ssl-cert* package is installed).

The certificate and related key are located here:

```
/etc/ssl/certs/ssl-cert-snakeoil.pem
/etc/ssl/private/ssl-cert-snakeoil.key
```

The names of the files are an indication that this pair is one that should inspire little confidence. The browser will then put up a warning about the certificate if it's being used for a server.

But they are perfectly fine for an initial attempt at configuration:

```
ServerName              localhost
ServerAdmin             root@localhost
ServerRoot              /apache
User                    www-data
Group                   www-data

ServerTokens            Prod
UseCanonicalName        On
TraceEnable             Off

Timeout                 5
MaxRequestWorkers       250

Listen                  127.0.0.1:80
Listen                  127.0.0.1:443

LoadModule              mpm_event_module        modules/mod_mpm_event.so
LoadModule              unixd_module            modules/mod_unixd.so
```

```
LoadModule              log_config_module       modules/mod_log_config.so

LoadModule              authn_core_module       modules/mod_authn_core.so
LoadModule              authz_core_module       modules/mod_authz_core.so

LoadModule              ssl_module              modules/mod_ssl.so

ErrorLogFormat          "[%{cu}t] [%-m:%-l] %-a %-L %M"
LogFormat               "%h %l %u [%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] \"%r\" %>s %b \
\"%{Referer}i\" \"%{User-Agent}i\"" combined

LogLevel                debug
ErrorLog                logs/error.log
CustomLog               logs/access.log combined

SSLCertificateKeyFile   /etc/ssl/private/ssl-cert-snakeoil.key
SSLCertificateFile      /etc/ssl/certs/ssl-cert-snakeoil.pem

SSLProtocol             All -SSLv2 -SSLv3
SSLCipherSuite          'kEECDH+ECDSA kEECDH kEDH HIGH +SHA !aNULL !eNULL !LOW !MEDIUM !MD5 !EXP !DSS \
!PSK !SRP !kECDH !CAMELLIA !RC4'
SSLHonorCipherOrder     On

SSLRandomSeed           startup file:/dev/urandom 2048
SSLRandomSeed           connect builtin

DocumentRoot            /apache/htdocs

<Directory />

        Require all denied

        Options SymLinksIfOwnerMatch
        AllowOverride None

</Directory>

<VirtualHost 127.0.0.1:80>

      <Directory /apache/htdocs>

        Require all granted

        Options None
        AllowOverride None

      </Directory>

</VirtualHost>

<VirtualHost 127.0.0.1:443>

        SSLEngine On

        <Directory /apache/htdocs>

            Require all granted

            Options None
            AllowOverride None

        </Directory>

</VirtualHost>
```

I won't be describing the entire configuration, only the directives that have been added since Tutorial 2. What's new is that in addition to port 80 we are also listening to *HTTPS port* 443. As expected, the *SSL* module is the new one to be loaded. We then configure the key and the certificate by using the *SSLCertificateKeyFile* and *SSLCertificateFile* directives. On the protocol line (*SSLProtocol*) it is very important for us to disable the older and insecure *SSLv2* protocol, but since the *POODLE* attack even *SSLv3* is also no longer secure. It would be best to permit only *TLSv1.2*, but not all browsers can handle it yet. So, we simply exclude *SSLv2* and *SSLv3* from use and for the time being are allowing TLSv1, very infrequent TLSv1.1 and

quantitatively dominating TLSv1.2. The handshake and encryption is done using a set of several algorithms. We use these cryptograph algorithms to define the *cipher suite*. It's important to use a clean *cipher suite*, because this is where snooping attacks typically take place: They exploit the vulnerabilities and the insufficient key length of older algorithms. However, a very limited suite may prevent older browsers from accessing our server. The proposed *cipher suite* has a high level of security and also takes into account some older browsers starting from Windows Vista. We are thus excluding Windows XP and very old versions of Android from communication.

The *HIGH* group of algorithms is the core of the *cipher suite*. This is the group of high encryption ciphers which *OpenSSL* provides to us via the *SSL module*. The algorithms listed in front of this keyword, which are also a part of the *HIGH* group, are given higher priority by being listed first. Afterwards we add the *SHA* hashing algorithm and exclude a number of algorithms that for one reason or another are not wanted in our *cipher suite*.

Then comes the *SSLHonorCipherOrder* directive. It is of immense importance. We often hear about *downgrade attacks* in SSL. This is when the attacker, a man-in-the-middle, attempts to inject himself into traffic and influence the parameters during the handshake in such a way that a protocol less secure than actually possible is used. Specifically, the prioritization defined in the *cipher suite* is defeated. The *SSLHonorCipherOrder* directive prevents this type of attack by insisting on our server's algorithm preference.

Encryption works with random numbers. The random number generator should be properly started and used, which is the purpose of the *SSLRandomSeed* directive. This is another place where performance and security have to be considered. When starting the server we access the operating system's random numbers in */dev/urandom*. While operating the server, for the *SSL handshake* we then use Apache's own source for random numbers (*builtin*), seeded from the server's traffic. Although */dev/urandom* is not the best source for random numbers, it is a quick source and also one that guarantees a certain amount of entropy. The qualitatively better source, */dev/random*, could in adverse circumstances block our server when starting, because not enough data are present, which is why */dev/urandom* is generally preferred.

We have also introduced a second *virtual host*. It is very similar to the *virtual host* for port 80. But the port number is *443* and we are enabling the *SSL engine*, which encrypts traffic for us and first enables the configuration defined above.

Step 2: Trying it out

For practice, as in the previous tutorials we have configured our test server for local IP address *127.0.0.1*. So let's give it a go:

```
$> curl -v https://127.0.0.1/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 443 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
* SSLv3, TLS handshake, CERT (11):
* SSLv3, TLS handshake, Server key exchange (12):
* SSLv3, TLS handshake, Server finished (14):
* SSLv3, TLS handshake, Client key exchange (16):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSL connection using ECDHE-RSA-AES256-GCM-SHA384
* Server certificate:
*    subject: CN=myhost.home
*    start date: 2013-10-26 18:00:21 GMT
*    expire date: 2023-10-24 18:00:21 GMT
* SSL: certificate subject name 'myhost.home' does not match target host name '127.0.0.1'
* Closing connection 0
* SSLv3, TLS alert, Client hello (1):
curl: (51) SSL: certificate subject name 'myhost.home' does not match target host name '127.0.0.1'
```

Unfortunately, we were not successful. It's no wonder, because we were talking to a server at IP address *127.0.0.1* and it replied to us with a certificate for *myhost.home*. A typical case of a handshake error.

We can instruct *curl* to ignore the error and open the connection nonetheless. This is done using the *--insecure*, or *-k* flag:

```
curl -v -k https://127.0.0.1/index.html
* Hostname was NOT found in DNS cache
*  Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 443 (#0)
* successfully set certificate verify locations:
*  CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
* SSLv3, TLS handshake, CERT (11):
* SSLv3, TLS handshake, Server key exchange (12):
* SSLv3, TLS handshake, Server finished (14):
* SSLv3, TLS handshake, Client key exchange (16):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSL connection using ECDHE-RSA-AES256-GCM-SHA384
* Server certificate:
*    subject: CN=myhost.home
*    start date: 2013-10-26 18:00:21 GMT
*    expire date: 2023-10-24 18:00:21 GMT
*    issuer: CN=myhost.home
*    SSL certificate verify ok.
> GET /index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 01 Oct 2015 07:48:13 GMT
* Server Apache is not blacklisted
< Server: Apache
< Last-Modified: Thu, 24 Sep 2015 11:54:56 GMT
< ETag: "2d-5207ce664322e"
< Accept-Ranges: bytes
< Content-Length: 45
<
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host 127.0.0.1 left intact
```

So, it works now and our SSL server is running. Admittedly with a lazy certificate and we are still far from use in production.

Below we will be discussing how to obtain an official certificate, how to install it correctly and how to tweak our configuration a bit.

Step 3a: Obtaining an SSL key and certificate

HTTPS adds an SSL layer to the familiar HTTP protocol. Technically, SSL (*Secure Socket Layer*) has been replaced by TLS (*Transport Security Layer*), but we still refer to it as SSL. The protocol guarantees encrypted and thus data traffic secured from eavesdropping. Traffic is encrypted symmetrically, guaranteeing greater performance, but in the case of HTTPS requires a public/private key setup for the exchange of symmetric keys by previously unknown communication partners. This public/private key handshake is done by using a server certificate which must be signed by an official authority.

Server certificates exist in a variety of forms, validations and scopes of application. Not every feature is really of a technical nature and marketing also plays a role. The price differences are very large, which is why a comparison is worthwhile. For our test setup we'll be using a free certificate that we will nonetheless have officially certified. Both can easily be obtained for a 12-month period for free at *StartSSL*. *StartSSL* is the subject of criticism, which is why this free service doesn't have a good reputation (e.g. there is a charge for revoking a certificate). It is however an easy way to get an official certificate for a test server and comparable free offers are for only 90 days or less.

This certificate is suitable for secure use on a production server, but nowadays it is probably better to use a more qualified *certificate authority* than the one proposed here. It has been announced that *Let's Encrypt* will be opening on November 16, 2015. It is a *CA* offering only free certificates. Once *Let's Encrypt* goes live, these instructions will be changed accordingly.

But for the moment we'll be using *StartSSL*. The provider first verifies the identity of the applicant and then, before issuing the certificate, checks his authorization to obtain a specific certificate for a specific domain. Verification takes place by an e-mail

sent to a predefined address for the desired certificate domain. Specifically, in the case of the domain *example.com*, this means that *StartSSL* sends an e-mail message with a security code to one of the three addresses: *postmaster@example.com*, *webmaster@example.com* or *hostmaster@example.com*. This prevents someone from getting a certificate for a third-party domain, because in this case he would not receive the message with the code.

These are currently the steps for obtaining a server certificate:

- Registration
- Verify private e-mail address
- Start creation of certificate and key
- Verify authorization for domain
- Complete creation of certificate and key
- Sign certificate
- Download and install signed certificate and key

It is of course also possible to create your own certificate and private key and only have the former signed online. This way the CA won't get its hands on our private key, which is highly recommended if not absolutely essential. What's important in both options is for the key to be protected by a strong password. We'll need this password when configuring the server later on.

Step 3b: Creating the certificate ourselves and having it officially signed

To generate a very good key we do the following:

```
$> openssl genrsa -des3 -out server.key 2048
```

Generating the key should take a moment, because a length of 2048 as specified is rather large and the necessary entropy must be found. It would also be possible to work with a length of 4096, but the little added cryptographic value comes at the cost of performance several magnitudes lower on the server. We can expect the request to proceed as follows:

```
Generating RSA private key, 2048 bit long modulus
.+++++++++++++++++++++++++++...
...
e is 65537 (0x10001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
```

Don't forget this pass phrase. We now use the new key to generate a *Certificate Signing Request*, or *CSR* for short:

```
$> openssl req -new -key server.key > server.csr
```

There are a few questions asked here that we answer in good conscience.

```
Enter pass phrase for server.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CH
State or Province Name (full name) [Some-State]:Bern
Locality Name (eg, city) []:Bern
Organization Name (eg, company) [Internet Widgits Pty Ltd]:example.com
Organizational Unit Name (eg, section) []:-
Common Name (eg, YOUR name) []:Christian Folini
Email Address []:webmaster@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:sjk3hrer8jk
An optional company name []:test company
```

We then receive a *CSR* named server.csr. This is what we take to *StartSSL* to have signed.

Step 4: Obtaining a certificate for the chain of trust

I will presume that you have obtained an officially signed certificate with associated key as described above or generated one yourself and have had it officially signed.

Becoming familiar with the inner workings of the *SSL/TLS protocol* can be quite demanding. A good introduction is the *OpenSSL Cookbook* by Ivan Ristić (refer to the links) or his extensive work *Bulletproof SSL and TLS*. One area that is hard to understand comprises the relationships of trust guaranteed by *SSL*. From the start the web browser trusts a list of certificate authorities, including *StartSSL*. When opening the *SSL* connection this trust is extended to our web server. This is done using the certificate. A chain of trust is created between the certificate authority and our server. For technical reasons there is an intermediate link between the certificate authority and our web server. We also have to define this link in the configuration. We first have to get the file:

```
$> wget https://www.startssl.com/certs/sub.class1.server.ca.pem -O startssl-class1-chain-ca.pem
```

When downloading I choose a file name a bit different from the one given. This gives us a bit more clarity for the configuration. The signed files are concatenated by the client during verification. Together the signatures on the certificates then create the chain of trust from our certificate to the *certificate authority*.

Step 5: Installing the SSL key and certificate

We now have the key and the two required certificates. Specifically:

- server.key, the *server key*
- server.crt, the *server certificate*
- startssl-class1-chain-ca.pem, the *StartSSL chain file*

We install them in two specially secured subdirectories in the configuration directory:

```
$> mkdir /apache/conf/ssl.key
$> chmod 700 /apache/conf/ssl.key
$> mv server.key /apache/conf/ssl.key
$> chmod 400 /apache/conf/ssl.key/server.key
$> mkdir /apache/conf/ssl.crt
$> chmod 700 /apache/conf/ssl.crt
$> mv server.crt /apache/conf/ssl.crt
$> chmod 400 /apache/conf/ssl.crt/server.crt
$> mv startssl-class1-chain-ca.pem /apache/conf/ssl.crt/
$> chown -R root:root /apache/conf/ssl.*/
```

Step 6: Automatically responding to the pass phrase dialog

When obtaining the key we had to define a pass phrase to unlock the key. In order for our web server to be able to use the key, we have to tell it about this code. It will then ask for it when the server starts. If we don't want that then we have to specify it in the configuration. To do this, we use a separate file that responds with the pass phrase when requested. We give this file the name */apache/bin/gen_passphrase.sh* and enter the pass phrase selected above:

```
#!/bin/sh
echo "S7rh29Hj3def-07hdkBgj4jDfg_skDg$48JuPhd"
```

This file must be specially secured and protected from prying eyes.

```
$> sudo chmod 700 /apache/bin/gen_passphrase.sh
$> sudo chown root:root /apache/bin/gen_passphrase.sh
```

Step 7: Configuring Apache

All of the preparations are now completed and we can do the final configuration of the web server. I won't be giving you the

complete configuration here, but only the specific server names and the tweaked SSL section:

```
ServerName      www.example.com

...

LoadModule              socache_shmcb_module    modules/mod_socache_shmcb.so

...

SSLCertificateKeyFile   conf/ssl.key/server.key
SSLCertificateFile      conf/ssl.crt/server.crt
SSLCertificateChainFile conf/ssl.crt/startssl-class1-chain-ca.pem
SSLPassPhraseDialog     exec:bin/gen_passphrase.sh

SSLProtocol             All -SSLv2 -SSLv3
SSLCipherSuite      'kEECDH+ECDSA kEECDH kEDH HIGH +SHA !aNULL !eNULL !LOW !MEDIUM !MD5 !EXP !DSS \
!PSK !SRP !kECDH !CAMELLIA !RC4'
SSLHonorCipherOrder On

SSLRandomSeed           startup file:/dev/urandom 2048
SSLRandomSeed           connect builtin

SSLSessionCache     "shmcb:/apache/logs/ssl_gcache_data(1024000)"
SSLSessionTickets   On


...


<VirtualHost 127.0.0.1:443>

    ServerName              www.example.com

...
```

It's also useful to enter the *ServerName* matching the certificate in the *VirtualHost*. If we don't do that, Apache will put up a warning (and then still select the only configured virtual host and continue to work correctly).

The *SSLSessionCache* and *SSLSessionTickets* options are new. These two directives control the behavior of the *SSL session cache*. The cache requires the *socache_shmcb* module, which provides caching functionality and is addressed using *mod_ssl*. It works as follows: During the SSL handshake the parameters of the connection such as the key and an encryption algorithm are negotiated. This takes place in public key mode, which is very CPU-intensive. Once the handshake is successfully completed, the server communicates with the client via higher performance symmetric encryption using the parameters that were just negotiated. Once the request has been completed and the *keep-alive* period in the new request has been exceeded, the TCP connection and the parameters imposed along with the connection are lost. If the connection is reopened just a short time later, the parameters will have to be negotiated once again. This is time-consuming, as we have just seen. It would be better if the parameters that were previously negotiated could be reactivated. This option exists in the form of the *SSL session cache*. This cache has traditionally been managed on the server side.

For a session cache via tickets, the parameters are combined in a session ticket and sent to the client, where it is stored on the client side, saving disk space on the web server. When opening a new connection the client sends the parameters to the server and it configures the connection accordingly. To prevent manipulation of the parameters in the ticket, the server temporarily signs the ticket and again verifies it when opening a connection. Something to consider in this mechanism is that the signature depends on a signing key and it is a good idea to regularly update the key that is for the most part dynamically generated. Restarting the server guarantees this.

SSL session tickets are recent and are now supported by all major browsers. They are also considered secure. However, this does not change the fact that there is a theoretical vulnerability in which session parameters are stolen on the client side.

Both types of session caches can be disabled. This is done as follows:

```
SSLSessionCache         nonenotnull
SSLSessionTickets       Off
```

Of course, this adjustment will have consequences in terms of performance. The loss of performance is however very small. It

would be surprising if a performance test would react to them being disabled with a performance drop of more than 10%.

Step 8: Trying it out

For practice, we have again configured our test server for local IP address *127.0.0.1*. To test if the certificate chain works, we can't just simply talk to the server using the IP address. We have to contact it using the specific host name. And this host name must of course match the one on the certificate. In the case of *127.0.0.1*, we do this by changing the host file in */etc/hosts*:

```
127.0.0.1   localhost myhost www.example.com
...
```

We can now access the URL https://www.example.com using a browser or curl. If this works without a certificate warning then we have configured the server correctly. The encryption and the chain of trust can be verified a bit more precisely by using the *OpenSSL* command line tool. Since *OpenSSL*, unlike the browser and curl, does not maintain a list of certificate authorities, we have to tell the tool what the certificate of authority is. We get it via *StartSSL*.

```
$> wget https://www.startssl.com/certs/ca.pem
...
$> openssl s_client -showcerts -CAfile ca.pem -connect www.example.com:443
```

Here, we instruct *OpenSSL* to use the built-in client, to show us the complete certificate information, to use the CA certificate just downloaded and to use these parameters to access our server. In the best case the output (slightly shortened) would look as follows:

```
CONNECTED(00000003)
---
Certificate chain
 0 s:/description=329817-gqai4gyx3JMxBbCV/C=CH/O=Persona Not Validated/OU=StartCom Free Certificate ...
   i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing/CN=StartCom Class 1 Primary ...
-----BEGIN CERTIFICATE-----
MIIHtDCCBpygAwIBAgIDArSFMA0GCSqGSIb3DQEBBQUAMIGMMQswCQYDVQQGEwJJ
...
...
...
x94JRF4camVVVDe3ae7TXZ/xl/Y8vR7TMbZJx4vg33IjnmLS6FOlf97BP6wA7wZN
zZnCQe+3NTU=
-----END CERTIFICATE-----
 1 s:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing/CN=StartCom Class 1 Primary ...
   i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing/CN=StartCom Certification ...
-----BEGIN CERTIFICATE-----
MIIGNDCCBBygAwIBAgIBGDANBgkqhkiG9w0BAQUFADB9MQswCQYDVQQGEwJJTDEW
...
...
...
p/EiO/h94pDQehn7Skzj0n1fSoMD7SfWI55rjbRZotnvbIIp3XUZPD9MEI3vu3Un
0q6Dp6jOW6c=
-----END CERTIFICATE-----
---
Server certificate
subject=/description=329817-gqai4fgt3JMxBbCV/C=CH/O=Persona Not Validated/OU=StartCom Free Certificate ...
issuer=/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing/CN=StartCom Class 1 Primary ...
---
No client certificate CA names sent
---
SSL handshake has read 4526 bytes and written 319 bytes
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : AES256-SHA
    Session-ID: FE496BB191B6888EA9CA3ED4E166707857186D5B32F1A0D9E418145D1B721CB4
    Session-ID-ctx:
```

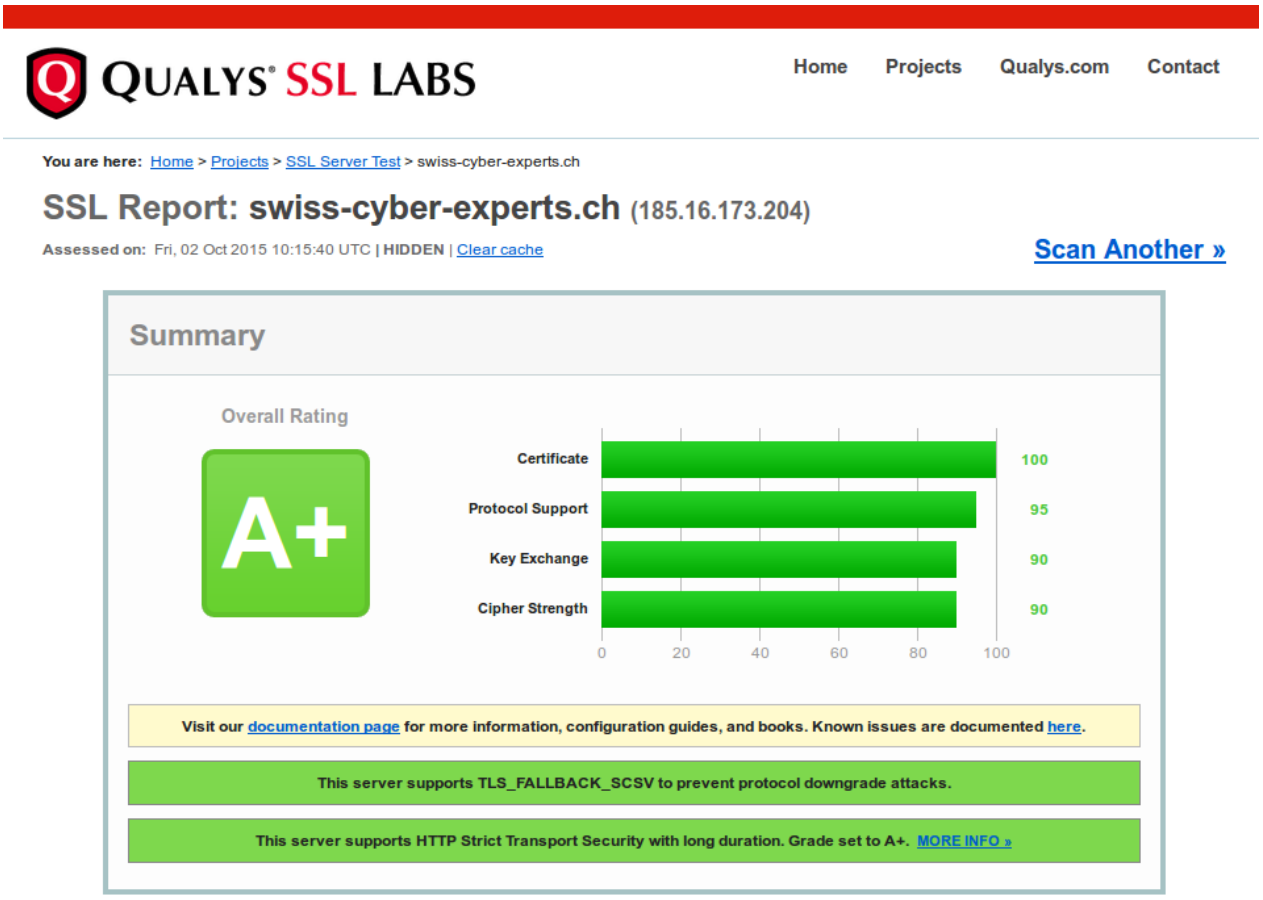
```
    Master-Key: 1BF16E22B0DF086E1AF4E13D9158AC0A3B1039E334C0C7F177A8757694B516E00E20AC3D6250B10D...
    Key-Arg   : None
    Start Time: 1294591828
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

We have completed configuring a clean *HTTPS server*.

Interestingly, there is something akin to a chart show for secure *HTTPS servers*. We'll take a look at it as an extra goodie.

Step 9 (Goodie): Checking the quality of SSL externally

Ivan Ristić, mentioned above as the author of several books on Apache and SSL, operates an analysis service for Qualys that checks *SSL web servers*. It is available at www.ssllabs.com. A web server configured like the one above earned me the highest grade of *A+* on the test.



The highest grade is attainable by following these instructions.

References

- Wikipedia OpenSSL
- Apache Mod_SSL
- StartSSL certificates
- SSLLabs
- OpenSSL Cookbook
- Bulletproof SSL and TLS
- Keylength.com – background information about ciphers and keys

License / Copying / Further use