2019 oplossingen labo 6 - Lars Lemmens

Met dank aan de Github van Martijn en natuurlijk Lars Lemmens

LABO 6

Check the contents of the ucll.be wildcard certificate, and make sure you understand the chain of trust.

```
'user@:~$' • echo | openssl s_client -connect ucll.be:443 | openssl x509 -text -noout
```

- x509 - is a certificate display and signing utility
- The -text argument prints out the certificate in text form.
- The -noout argument prevents output of the encoded version of the request

We'll start this lab by making our very own SSL certificate. Before we can create a certificate we need to generate an RSA private key.

```
'user@:~$' • openssl genrsa -aes128 -out TomBola.keys 2048
```

- The genrsa command generates an RSA private key

Use the following command to remove the pass phrase:

```
'user@:~$' • openssl rsa -in TomBola.keys -out TomBola.keys
```

- The -in command specifies the input filename to read a request from or standard input if this option is not specified
- The -out command specifies the output filename to write to or standard output by default.

Now create the Certificate Signing Request.

```
'user@:~$' • openssl req -new -key TomBola.keys -out TomBola.csr
```

Lazy admins will probably prefer the following command, its result is the same as all three commands above combined:

```
'user@:~$' • openssl req -new -newkey rsa:2048 -keyout TomBola.keys -nodes -out TomBola.csr
```

- The -new command is used to generate a new certificate request
- The -newkey command creates a new certificate request and a new private key
- The -keyout command gives the filename to write the newly created private key to.
- The -nodes command is used if a private key is created it will not be encrypted.
- The -out command specifies the output filename to write to or standard output by default.
- The nodes argument -nodes tells OpenSSL not to encrypt the private key

This verification step is optional but recommended. Now we need somebody to sign our key. We will use our own private key for the signature, creating a self-

signed certificate.

As a self-signed certificate is signed by itself it won't be trusted by client software like email clients and web browsers.

Only public keys of real Certificate Authorities (CA) are included in the browsers or OSs certificate stores.

```
'user@:~$' • openssl x509 -req -days 365 -in TomBola.csr -signkey TomBola.keys -out TomBola.crt
```

- The -req command is by default a certificate is expected on input. With this option a certificate request is expected instead.
- The -signkey command causes the input file to be self signed using the supplied private key.
- The -days command is used (when the -x509 option is being used) specifies the number of days to certify the certificate for.

If somehow you forgot which key/crt/csr files belong to each other, compare their modulus values. They should be the same. To ease the comparing process create a hash value first.

```
'user@:~$' • openssl x509 -in TomBola.crt -noout -modulus | md5sum; \
openssl req -in TomBola.csr -noout -modulus | md5sum; \ openssl rsa -in TomBola.keys -noout -modulus |
 md5sum
```

- The -in command specifies the input filename to read a certificate from or standard input if this option is not specified.
- The -noout command is used to prevent output of the encoded version of the request
- The -modulus command prints out the value of the modulus of the public key contained in the request.
- The -days command is used (when the -x509 option is being used) specifies the number of days to certify the certificate for.

As we need a Subject Alternative Name (SAN) field to specify additional names associated with our certificate the first step is to create a small openssl configuration file.

```
FILE: uclllabs_be.conf

[ req ]
default_bits = 4096
prompt = no
encrypt_key = no
default_md = sha512
distinguished_name = dn
req_extensions = v3_req

[ dn ]
C = BE
O = UC Leuven
CN = *.uclllabs.be

[ v3_req ]
subjectAltName = DNS:uclllabs.be
```

In the next step, we generate our CSR

```
'user@:~$'
 • openssl req -new -newkey rsa:4096 -nodes -out star_uclllabs_be.csr -keyout star_uclllabs_be.key -subj "
/C=BE/ST=/L=/O=UC Leuven/CN=*.uclllabs.be" -sha512 -config uclllabs_be.cnf
'user@:~$' • openssl req -new -nodes -out star_uclllabs_be.csr -keyout star_uclllabs_be.key -config uclllabs_be.cnf
```

- The command req is a certificate request and certificate generating utility
- The -new argument generates a new certificate request
- The -newkey argument creates a new certificate request and a new private key.
- The nodes argument -nodes tells OpenSSL not to encrypt the private key
- The -out argument is specified then if a private key is created it will not be encrypted.
- The -keyout argument gives the filename to write the newly created private key to.
- The -subj argument replaces subject field of input request with specified data and outputs modified request.
- The -config argument allows an alternative configuration file to be specified

First check whether the certificate's common name corresponds with the server name, check the expiration date and check the chain of trust.

```
'user@:~$' • echo | openssl s_client -connect ucll.be:443 | openssl x509 -text -noout | grep -A2 Validity
```

- x509 - is a certificate display and signing utility
- The -text argument prints out the certificate in text form.
- The -noout argument prevents output of the encoded version of the request

Second, verify the certificate is not listed on the included CRL (Certificate Revocation List):

http://crl3.digicert.com/TERENASSLCA3.crl

```
'user@:~$' 1) wget http://crl3.digicert.com/TERENASSLCA3.crl
'user@:~$' 2) openssl crl -in TERENASSLCA3.crl -text -noout
```

- The -in argument specifies the input filename to read from or standard input if this option is not specified
- The -text argument prints out the CRL in text form.
- The -noout argument doesn't output the encoded version of the CRL

Unfortunately, this does no seem to work. OpenSSL stops and spawns an error message

The reason is that TERENA published their CRL in DER format, and by default OpenSSL uses the PEM format. Lets convert the CRL to the correct format and try again:

```
'user@:~$' 1) openssl crl -inform DER -in TERENASSLCA3.crl -outform PEM -out TERENASSLCA3.crl.pem
'user@:~$' 2) openssl crl -in TERENASSLCA3.crl.pem -text -noout
```

- The -inform argument specifies the input format
- The -in argument specifies the input filename to read from or standard input if this option is not specified
- The -outform argument specifies the output format
- The -out argument specifies the output filename to write to or standard output by default.

The last two commands could be combined in one so the conversion from DER to PEM format is not necessary anymore:

```
'user@:~$' • openssl crl -inform DER -in TERENASSLCA3.crl -text -noout
```

And without first downloading the crl:

```
'user@:~$' • wget -q -O - http://crl3.digicert.com/TERENASSLCA3.crl | openssl crl -inform DER -text -noout
```

And with extracting the CRL URI automatically:

```
'user@:~$' • wget -q -O - $(echo | openssl s_client -connect ucll.be:443 2>/dev/null | openssl x509 -text -noout |
 grep -oP 'URI:\K.*\.crl'| head -1) | openssl crl -inform DER -text -noout
```

- The wget command is a non-interactive network downloader
- The -q command is used to turn off Wget's output
- The -o command is specified for the outpout document
- The -text argument prints out the CRL in text form.
- The -noout argument doesn't output the encoded version of the CRL

Let's find the serial number of our certificate:

```
'user@:~$' • openssl x509 -in cert.pem -text
```

shows our certificate and we find the certificate serial number being: "0d:d9:43:24:0a:84:a5:e6:36:94:ff:c7:eb:e8:1e:3f"

Or in one command with:

```
'user@:~$' • echo | openssl s_client -connect ucll.be:443 2>/dev/null | openssl x509 -text -noout |
 grep -A1 -i serial
```

- The -text argument prints out the CRL in text form.
- The -noout argument doesn't output the encoded version of the CRL
- The -A prints NUM lines of trailing context after matching lines.
- The -i command is used to ignore case

If we would like to verify this serial number manually we need to remove the colons or add colons to the serials in the crt.

Removing should be a peace of cake since the previous labs. Adding a colon every two characters is a bit more complex: "

```
'user@:~$' • echo 0d:d9:43:24:0a:84:a5:e6:36:94:ff:c7:eb:e8:1e:3f | tr -d ':'
=> OUTPUT: 0dd943240a84a5e63694ffc7ebe81e3f

'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/..\B/&:/g'
=> OUTPUT: 0d:d9:43:24:0a:84:a5:e6:36:94:ff:c7:eb:e8:1e:3f
```

- The .. mean: look for exactly two characters.
- &: means: replace these two dots (their corresponding found characters) itself and add a colon. Thus ab becomes ab:
- The g at the end tells sed not to stop on the first occurrence
- And finally: /B makes sure that the .. only matches at the word boundary.

```
'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/..//'
=> OUTPUT: d943240a84a5e63694ffc7ebe81e3f

'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/../&/'
=> OUTPUT: 0dd943240a84a5e63694ffc7ebe81e3f

'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/../&:/'
=> OUTPUT: 0d:d943240a84a5e63694ffc7ebe81e3f

'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/../&:/g'
=> OUTPUT: 0d:d9:43:24:0a:84:a5:e6:36:94:ff:c7:eb:e8:1e:3f:
```

```
'user@:~$' • echo 0dd943240a84a5e63694ffc7ebe81e3f | sed 's/..\B/&:/g'
=> OUTPUT: 0d:d9:43:24:0a:84:a5:e6:36:94:ff:c7:eb:e8:1e:3f
```

Of course, much easier is to just use the -serial parameter of OpenSSL, it will be displayed in the correct format automatically:

```
'user@:~$' • openssl x509 -in cert.pem -serial -noout
```

- The -noout argument doesn't output the encoded version of the CRL
- The -serial command outputs the certificate serial number.

Of course it is even more easy to just let OpenSSL do all the hard work, just like we do with our web browsers and mailclients. It is the client software which verifies the certificate and not the user.

```
'user@:~$'
 • openssl verify -CAfile TERENASSLCA3.crl.pem -crl_check cert.pem cert.pem: C = BE, ST = Vlaams-Brabant, L = L
*.ucll.be
```

- The verify command is used to verify certificate chains
- The -CAfile command is a file of trusted certificates.
- The -crl_check command checks end entity certificate validity by attempting to lookup a valid CRL

Now its time to do something with our newly created certificate (TomBola.crt).

OpenSSL not only includes SSL/TLS client software but also a server, which can act as a very basic web server:

```
'user@:~$' • openssl s_server -accept 10000 -cert TomBola.crt -key TomBola.key -www -state
```

- The -www command sends a status message back to the client when it connects.
- The -state command prints out the SSL session states.
- The -cert command is to specify which certificate to use
- The -state argument is used to display the various SSL states and messages used to build the secure connection

```
'user@:~$' • openssl s_client -connect localhost:10000
```

Use the private key from your certificate with the following command-line

```
'user@:~$' • openssl dgst -sha256 -sign TomBola.key -out examenvragen.pdf.sha2 examenvragen.pdf
```

- The dgst function outputs the message digest of a supplied file or files in hexadecimal form. They can also be used for digital signing and verification.
- The -sign command digitally signs the digest using the private key in "filename".

When you receive the document it's signature can be verified with the following command:

```
'user@:~$' • openssl dgst -sha256 -verify TomBola.pub -signature examenvragen.pdf.sha2 examenvragen.pdf
```

- The dgst function outputs the message digest of a supplied file or files in hexadecimal form. They can also be used for digital signing and verification.
- The -signature command is used to verify the actual signature

Use the following OpenSSL command to extract the public key from the certificate and execute the command above to verify the integrity of the document and also authenticate its sender/creator.

```
'user@:~$' • openssl x509 -in TomBola.crt -pubkey -noout > TomBola.pub
```

Use Wireshark to analyze and observe the TLS handshake between your browser and a https site of your choice. Try to answer the following questions:

Which version of the SSL/TLS protocol is used?

Which signature hash algorithms are advertised as supported by your browser?

Which cipher suites are supported by the client (browser), and which are supported by the server?

Which cipher suite is chosen? Explain the different parts of the cipher suite.

Is it the client or the server who defines the cipher suite to be used?

If you need to see the contents of SSL/TLS encrypted packets you need to feed the correct decryption keys into Wireshark.

If you have access to the private key and no Diffie-Hellman is used to calculate the session keys (shared secrets), Wireshark can decrypt all data.

In most cases you do not have access to the private key or DHE is used to calculate the session keys, thus the above option will not work.

Fortunately, some browsers (like firefox) have the ability to save these session keys to a file, allowing Wireshark to decrypt all SSL/TLS data to/from any website:

```
'user@:~$' 1) export SSLKEYLOGFILE=/home/slimmerik/sslkey.log
'user@:~$' 2) firefox &
'user@:~$' 2) sudo wireshark -o ssl.keylog_file:/home/slimmerik/sslkey.log
```

```
# crl □          Certificate Revocation List (CRL) Management.
# rsa □           RSA key management (private key).
# csr □           Certificate Signing Request (CSR) Management.
# x509 □         X.509 Certificate Data Management (public key certificate).
# s_client □    Openssl SSL/TLS client (e.g. an https client)
# s_server □    Openssl SSL/TLS server (e.g. an https server)
# enc □           Encoding with Ciphers (encryption and decription).
```

Exercise 1:

Take a close look at the contents of cert.pem. It is a pem encoded digital certificate.

With the following OpenSSL magic the certificate can be displayed in a more

human-friendly format."

```
'user@:~$' • openssl x509 -text -noout -in cert.pem
```

Exercise 2:

Let's say we would like to check the expiration date of the certificate used on https://wiki.uclllabs.be. We could use the following set of commands:"

```
'user@:~$' • openssl s_client -connect wiki.uclllabs.be:443 > tempfile
```

Now open the temp file with vim, your preferred text editor and delete all lines before -----BEGIN CERTIFICATE----- and after -----END CERTIFICATE-----.

Next use the command below to find the correct expiration date:

```
'user@:~$' • openssl x509 -noout -enddate -in tempfile
```

And if you managed to understand everything this far in this lab, you should

be able to glue the above together in just one command line.

```
'user@:~$' • echo | openssl s_client -connect wiki.uclllabs.be:443 2>/dev/null | perl -nle 'print if /BEGIN/../END/' |
 openssl x509 -noout -enddate
'user@:~$' • echo | openssl s_client -connect wiki.uclllabs.be:443 2>/dev/null | openssl x509 -noout -enddate
```

- The -enddate command prints out the expiry date of the certificate, that is the notAfter date.

Now with a few additions, the above could be used as a base for a fully automated certificate expiration checker

Create a textfile with all your domains in it for which ssl certificates are used. Of course this could be automated too.

There are of course less ugly ways for achieving the same

```
'user@:~$' • for foo in $(cat hosts); do if [[ $(echo $(date -d"$(echo | openssl s_client -connect $foo:443 2>
/dev/null | openssl x509 -noout -enddate
| cut -d'=' -f2 | awk '{print $2 " " $1 " " $4}')" +%s) - $(date +%s) | bc) -gt 0 ]]; then echo $foo: OK; else echo
$foo: EXPIRED;fi;done
```

And now a shorter version with some more OpenSSL magic

```
'user@:~$' • for foo in $(cat hosts); do if echo | openssl s_client -connect $foo:443 2>/dev/null
| openssl x509 -noout -checkend 0; then echo $foo: OK; else echo $foo: EXPIRED; fi;done
```

- The -checkend command checks if the certificate expires within the next arg seconds and exits non-zero if yes it will expire or zero if not.

And magic++:

```
'user@:~$' • for foo in $(cat hosts); do check_ssl_cert -H $foo;done
```

- The check_ssl_cert checks the validity of X.509 certificates
- The -H command is used to specify the host

Exercise 3:

Create a oneliner which shows the amount of currently revoked certificates in the Terena SSL CA revocation list.

Do not create temporary files. The CRL can be found at

http://crl3.digicert.com/TERENASSLCA3.crl

."

```
'user@:~$' • wget -q -O - http://crl3.digicert.com/TERENASSLCA3.crl | openssl crl -inform DER -text -noout |
 grep -P '^\s+Serial Number' | wc -l
```

- The -inform command specifies the input format normally the command will expect an X509 certificate but this can change if other options such as -req are present.

Exercise 4:

Companies like Google and Microsoft make heavily use of the X.509 subjectAltName extension.

UCLL also uses this extension to add an alternative name *.ucll.be to the common name (ucll.be) of the certificate.

Create a oneliner which calculates the amount of DNS Subject Alternate Names used in the SSL certificate of gmail.com."

```
'user@:~$' • echo | openssl s_client -connect facebook.com:443 2>/dev/null | openssl x509 -text -noout | grep -o 'DNS:' | wc -l
```

Exercise 5:

Create a CLI oneliner to find a match between different rsa private key files and their companion crt files. The output should look something like:

```
alfa.key matches beta.crt
gamma.key matches delta.crt
```

First we need to generate some certificates so we can actually verify our solution. The following command creates a self-signed certificate in one step:

```
'user@:~$'
 • openssl req -x509 -sha256 -newkey rsa:2048 -keyout tombola_uclllabs_be.key -out tombola_uclllabs_be.crt -day
"/C=BE/ST=/L=/O=UC Leuven/CN=tombola.uclllabs.be"
```

- The -subj command outputs the "hash" of the certificate subject name.

With the following for loop we can easily generate a few certificates to test with:

```
'user@:~$' • for foo in TomBola GreetSchap VanderNeffe LukRaak AlainProvist; do
 openssl req -x509 -sha256 -newkey rsa:2048 -keyout
"$foo"_uclllabs_be.key -out "$foo"_uclllabs_be.crt -days 1024 -nodes -subj
```

As explained in this lab, the modulus of corresponding files (CSR, private KEY and Certificate) should be the same. We'll create a oneliner which tests wich files belong to one another. I.e. which files have the same modulus.

```
'user@:~$' • for key in $(ls -1 *.key); do for crt in $(ls -1 *.crt); do if [[ $(openssl rsa -in $key -noout -modulus | md5sum) == $(openssl x509 -in $crt -noout -modulus | md5sum) ]]; then echo $key matches $crt;fi ;done;done
```

Or a slightly shorter version:

```
'user@:~$' • for key in $(ls *.key); do for cert in $(ls *.pem); do [[ $(openssl rsa -in $key -noout -modulus | md5sum) == $(openssl x509 -in $cert -noout -modulus | md5sum) ]] && echo $key matches to $cert;done;done
```

Exercise 6:

Create a CLI oneliner using openssl to retrieve the certificate of the server wiki.uclllabs.be and to encrypt the text

'Lets make CNW2 great again'" with it’s public key. In the first step, we willll

connect to https://wiki.uclllabs.be and extract the public key from the presented certificate:"

```
'user@:~$' • echo | openssl s_client -connect wiki.uclllabs.be:443 2>/dev/null | openssl x509 -noout -pubkey > public.pem
```

Now the public key is saved in a tempfile we can use it to encrypt arbitrary data.

```
'user@:~$' • echo "Let's make CNW2 great again" |
 openssl rsautl -encrypt -pubin -inkey public.pem -out CNW2.encrypted
```

- The rsautl command can be used to sign, verify, encrypt and decrypt data using the RSA algorithm.
- The -encrypt command encrypts the input data using an RSA public key.
- The -pubin command is used to input file is an RSA public key.
- The -out command specifies the output filename to write to or standard output by default.

Or without first downloading the public key:

```
'user@:~$' • echo "Let's make CNW2 great again" | openssl rsautl -encrypt -pubin -inkey <(echo |
 openssl s_client -connect wiki.uclllabs.be:443 2>/dev/null | openssl x509 -noout -pubkey) -out CNW2.encrypted
```

Exercise 7:

Experiment with the tools sslyze and nmap to enumerate supported ssl cipher suites for various websites. Make sure you understand its output. See the folloing example:

```
- sslyze --regular --http_headers wiki.uclllabs.be
- nmap --script ssl-enum-ciphers -p 443 wiki.uclllabs.be
```

Exercise 8:

Let’s Encrypt has rate limits which prevent issuing more than 20 new certificates per domain per week. They use the public suffix list for this.

The limit is per registered domain, not per subdomain. So all student certificates count for the same uclllabs.be suffix.

```
"root #" • certbot --apache -d "tom.x.cnw2.uclllabs.be, bola.x.cnw2.uclllabs.be, tom.bola.x.cnw2.uclllabs.be"
```

- The certbot command is used to obtain and install HTTPS/TLS/SSL certificates
- The --apache command is used to use the Apache plugin for authentication & installation
- The -d command is used for domain names to apply

Exercise 9:

Create a CLI oneliner using OpenSSL to retrieve the certificate of the server wiki.uclllabs.be and to display only its fingerprint, serial and public key. Sample output:"

```
'user@:~$' • echo | openssl s_client -connect wiki.uclllabs.be:443 2>/dev/null |
 openssl x509 -noout -fingerprint -serial -pubkey
```

Revision #1
Created 17 June 2021 14:15:19 by Jasper G.

Updated 3 December 2021 22:13:09 by Jasper G.