LibreSSL/libtls

Petr Ročkai

What is LibreSSL?

- a fork of OpenSSL
- came about soon after heartbleed
- removed a lot of old code
- it also contains a new TLS API

Getting and Building

- it’s recommended to work on `aisa.fi.muni.cz`
- fetch and unpack sources from `https://libressl.org`
- `./configure --prefix=$HOME/libressl`
- `make && make install`

Using `libtls`

- `#include <tls.h>`
- use appropriate `-I` and `-L` flags
- set `LD_LIBRARY_PATH` or use `-static`
- the docs are online (man pages)

The Context

- declared as `struct tls *ctx`
- 2 initialisation functions:
  - `tls_client()`
  - `tls_server()`
- `tls_accept_socket` makes a new context

Configuration

- `tls_config_new()` creates a `tls_config`
- `tls_config_free()` destroys it
- `tls_config_set_cert_file( conf, "cert.pem" )`
- `tls_config_set_key_file( conf, "key.pem" )`
- `tls_config_set_ca_file( conf, "roots.pem" )`
- `tls_configure( context, conf )`

Connecting as a Client

- use `tls_connect( context, host, port )`
- this sets up the low-level sockets
- you can talk to the server afterwards:
  - `tls_read( context, buf, len )`
  - `tls_write( context, buf, len )`
- the SSL handshake is done automatically

Setting up a Server

- you have to set up the sockets yourself
- use `tls_accept_socket` to set up SSL
  - give it the `fd` you got from `accept`
  - it creates a new context
- proceed with `tls_read` and `tls_write`
  - be sure to use the newly created context

Cleaning Up

- `tls_close()` closes the SSL session
- `close()` shuts down the socket
- `tls_free()` frees up memory
- clean up all contexts you create

Homework

- you hand in 2 files: `client.c` and `server.c`
- put them in a zip file
- these are the only `.c` files you can use
- you can include a common header if you want

Homework: Server

- write a simple SSL server
- invocation: `./server port cert.pem key.pem`
- establish an SSL connection with a client
- read 4 bytes from the client
  - if they read `ping`, send back `pong`

Creating a Socket

```c
int sockfd = socket( AF_INET, SOCK_STREAM, 0 );
if ( sockfd < 0 )
    return perror( "socket" ), 1;
```

Binding a Socket

```
struct sockaddr_in sa;
sa.sin_family = AF_INET;
sa.sin_port = htons( 9000 );
sa.sin_addr.s_addr = INADDR_ANY;

if ( bind( sockfd, &sa, sizeof( sa ) ) )
    return perror( "bind" ), 1;
```

Accepting Connections

```
if ( listen( sockfd, 5 ) )
    return perror( "listen" ), 1;
if ( ( fd = accept( sockfd, NULL, NULL ) ) < 0 )
    return perror( "accept" ), 1;
/* fd is the connected socket */
```

Homework: Client

- write a simple SSL client
- invocation: `./client host port [trusted.pem]`
- `trusted.pem` contains trusted (root) certs
  - if not given, use the system default
- connect to `host` at port `port`
- send the string `ping` and wait for any response

Homework: Client Output

- print the issuer and the subject of the server cert
- print the cipher that's being used on your connection
- print the response you obtain from the server

```
issuer: <string>
subject: <string>
cipher: <string>
response: <string>
```

Homework: Various

- the client must validate the server cert
- the server only needs to serve 1 client at a time
- use `tls_error` to inform about problems
- absolutely no buffer overflows allowed
- deadline: 29.11. midnight

Homework Testing

- you can use `openssl` to test both parts
- use `s_server` to test your client
- use `s_client` to test your server
- e.g. `openssl s_client -host localhost -port 9000`
- your client should be able to talk to your server

Certificates for Testing

- you will need to generate a CA certificate
  - the CA cert can be self-signed
- you can use `certtool` to make the certs
  - `certtool -p --outfile ca.key`
  - `certtool -s ...`
  - `certtool -c ...`