Computer Networks *(Redes de Computadores - RCOMP)* – 2024/2025

Laboratory Class 14 (PL14 – 3 hours)

Implementing a generic TLS client to test TCP servers. Securing network applications with SSL/TLS.

1. SSL/TLS - Secure Sockets Layer/Transport Layer Security

SSL/TLS is a protocol designed to enforce security in network communications at layer four, meaning over UDP and over TCP, so, unlike IPsec operating at layer three, it's directly used by network applications.

Due to the need of negotiating and establishing security parameters between two applications, SSL/TLS is mostly used over TCP connections using the client-server model.

For connection-less protocols like UDP, specific issues arise and there's a specific implementation known as **Datagram Transport Layer Security** (DTLSv1.0 and DTLSv1.2). For VPN and tunnel implementations, DTLS has the advantage of not suffering from the *TCP meltdown* problem, and it's used for instance by *OpenVPN*.

We will focus on SSL/TLS over TCP only, there are several main versions that have evolved along the time, by chronological order: SSLv2; SSLv3; TLSv1.0; TLSv1.1; TLSv1.2; TLSv1.3. After version 3, SSL was renamed to TLS. SSL named versions are now regarded as less safe and shouldn't be used.

By using the SSL/TLS protocol, standard non-secure application protocols like HTTP are converted to secure protocols without any change on the standard application protocol itself, they are often renamed with an S suffix (e.g., HTTPS). SSL/TLS is enforced once the TCP connection is established, prior to any information exchange regarding the standard protocol to be secured.

Immediately after the TCP connection establishment, SSL/TLS protocol messages are exchanged between enrolled nodes, the client sends a **TLS Client Hello** message, and the server replies back with a **TLS Server Hello** message. This handshake establishes two things: the **TLS version** to be used and the **cypher suite** to be used.

Depending on the selected cypher suite, if it requires public key certificates for authentication, they will be also exchanged. In that case, the server will always send a certificate, and it may demand, or not, a certificate from the client.

Once the TLS handshake has ended, then the standard protocol to be secured can starts its own data exchanges, in the case of HTTP the client sends a HTTP request, and the server replies back with a HTTP response. However, these transactions are now secured by TLS, thus both the client and server will no longer use the standard read/write functions, instead they use specific function provided by the SSL/TLS API, in our case, it will be OpenSSL.

1.1. First of all, authentication

The SSL/TLS protocol provides **authentication**, **privacy** (confidentiality), and **integrity**, each of these features is implemented by a cryptographic function/algorithm, and they are all represented by names in each cypher suite.

Ultimately, when the TLS handshake is finished, and the control is passed to the standard application protocol to be secured, a secret key for a symmetrical key cypher has been established and it's known only by the two enrolled nodes.

Achieving such a goal without being absolutely sure we are talking with the correct counterpart application is pointless, we could end up exchanging encrypted information with a node that can decrypt it.

To authenticate enrolled applications, SSL/TLS provides two alternatives:

- **PSK (Pre-Shared Key)**: this is rather similar to users' authentication by password. Is assumes there's a secret both applications known, and nobody else knows, in SSL/TLS it's a cryptographic key and not a password, but it works the same. If one application sends data encrypted with that secret key, and the counterpart can decrypt it, then we must assume it's authentic because we also assume nobody else knows the secret key. Of course, PSK requires that both nodes are under control of the same administrator such as he is able to manually place the secret key on both sides. So, for anonymous clients contacting public servers on the internet, PSK is not an option.

- **Public Key Certificates**: when using asymmetrical key cyphers, data is encrypted with a public key, and it can only be decrypted with the matching private key. Unlike with PSK, private keys are never shared, thus are much safer. If one application sends data encrypted with the counterpart's public key, and the counterpart can decrypt it, then we must assume it's authentic, because we know only that counterpart has the matching private key. Yet, the sending application has be sure it's using the intended counterpart's public key, and that's what **public key certificates** are for.

1.2. Public Key Certificates

Public Key Certificates are used to authenticate public keys, they contain a public key and an entity identification (subject). The certificate is digitally signed by another entity (issuer), unless it's a self-signed certificate (issuer=subject). The signature in the certificate, may be checked by using the issuer's public key, which in turn will be also available on another public key certificate.

So, from a public key certificate, a chain of certificates starts, going from issuer to issuer.

Applications only trust certificates belonging to issuers they know, the corresponding certificates must be provided to the application, in Linux systems, they are usually stored in a folder (e.g., /etc/ssl/certs).

And yet, the trust is transitive, meaning if the issuer's certificate was issued by a trusted issuer, then it's trusted as well.

So, to check if a certificate is trusted, the chain path is followed, if it hits a trusted certificate, then is trusted. This is why, when during the TLS handshake, an application sends the counterpart its certificate, it should as well send the full chain of certificates starting from there. The counterpart may not have all certificates required to follow the chain.

1.3. Authenticating public servers

Regarding authentication with public key certificates, at least two different scenarios can be established. One case we will describe first is for a public server on the internet to be accessed by many anonymous clients.

Basically, this means the server has no previous knowledge about the clients' existence, and the only thing clients known about the server is its DNS name.

The server couldn't care less about the client's authenticity, it's supposed to be publicly available to all clients, and thus it will not demand for a client's certificate. The server may require later user authentication, but for that it will use the standard application protocol being secured (e.g., HTTP).

From the point of view of the client, however, things are different, the client must be sure it's talking with the server it desires, and not a fake one.

For public key certificates to become useful in this scenario, they must contain the DNS name of the server they are issued for. Of course, the logical place to put that is the subject field of the certificate.

The subject field is a text with several attributes, each in the form:

/ATTRIBUTE-NAME=VALUE

It has been established the common name (CN) attribute must match the DNS name of the server, even though the DNS name may be also placed in extension elements of the certificate, it should always be in the CN attribute of the certificate's subject. At least for a certificate belonging to a server with a DNS name.

So, when contacting https://www.dei.isep.ipp.pt, the received certificate should have in the subject field (among other attributes):

/CN=www.dei.isep.ipp.pt

This is one additional check clients must do, nevertheless, to start with, the certificate must be trusted. To be trusted, it must be in a certification chain of a trusted certificate (trusted CA). The list of worldwide trusted Certification Authorities is part of operating systems, and client applications, like web browsers, these lists are also updated together with operating systems and client applications.

Worldwide trusted Certification Authorities are the only ones that can provide a certificate for a server, such as standard client applications and operating systems will trust it. When assigning a certificate, one main fact to be analysed by the issuing CA, is if the requestor is the administrator of the server matching the requested DNS name.

There are some trusted Certification Authorities, supplying free public key certificates, as far as the administrative ownership of the server with the requested DNS name is demonstrated. One of those, is Let's Encrypt (https://letsencrypt.org/).

1.4. Authenticating application instances

A different, but common scenario, is when there's a set of fixed, well known, application instances running, and mutual authentication is required.

For this scenario, self-signed certificates are a solid solution. Each application instance will have its own self-signed public key certificate, and the matching private key.

To make it clear, **private keys are an absolute secret**, the private key owner (in this case the application instance) will never, in any case, allow anybody to see the private key. This is an advantage of asymmetric key cyphers; private keys are never shared.

For this scenario, during the TLS negotiation, **the server instance should always demand a client's certificate** (unless clients' authentication is not required). By default, servers don't demand a client's certificate.

In each application instance, the only certificates to be trusted are those belonging to other trusted application instances. This means, an access by an application providing a not included certificate, will not be authorized. Applications providing an included certificate, are allowed, and at the same time authenticated.

Also, in this case, the subject may basically ignored. Nevertheless, certificates belonging to different instances should have different subjects representing those instances. Depending on the used SSL/TLS implementation and API, when several certificates are loaded by the application, they are often organized by the subject's content, if two certificates have the same subject, they will collide and one of them will not get loaded.

Unlike what would happen if PSK was used, each application instance may have an independent administrator. Each will create a self-signed public key certificate and the matching private key, would store the private key in an inaccessible place, and then would send the public key certificate all other administrators, so they could grant access by adding it to the trusted list.

Protecting private keys is very important, they are never transmitted, but they have to be stored somewhere, usually a file. The underling file system permissions can be used to protect the file, nobody except one required user should be able to access it. Of course, when the application that needs the private key starts, it must be running as that required user.

One other, often used, method to protect files holding private keys is by encrypting those files with a provided password. The big inconvenience of this approach is that the application that needs the private key has to know the password to decrypt the file. This means the password has to be either hardcoded into the application or stored in some configuration file within the system.

1.5. OpenSSL library

OpenSSL (https://www.openssl.org/) is a library implementing SSLv2, SSLv3, TLSv1.0, TLSv1.1, TLSv1.2, TLSv1.3, DTLSv1.0, and DTLSv1.2. Offers a C language API and is standard on most Linux distributions. The API offers a wide range of cryptographic functions, and SSL/TLS related functions.

It’s available for several platforms, not only for the UNIX family, but also for Windows and OS X. The API is for C language, but there are third party wrappers to a wide range of languages (e.g., Python, NodeJS, PHP, Perl).

The package also includes the **openssl** command line utility, with several uses like for instance generating public and private keys, and public key certificates. The default format to save and load certificates and private keys is PEM (Privacy Enhanced Mail), uses base64 representation, so it’s actually text.

1.6. Java Secure Socket Extension (JSSE)

Java has its own cryptographic implementations, the Java Cryptography Architecture (JCA), supporting a wide range of algorithms and security protocols. It includes the Java Secure Socket Extension (JSSE) dedicated to the SSL/TLS support, one of the packages is **javax.net.ssl**.

The **javax.net.ssl** package contains a number of classes very similar to those in **java.net**, we already know, for instance, SSLSocket, and SSLServerSocket.

The Java environment also has a command line utility, named **keytool**, used to manage certificates and private keys. With this utility, certificates and private keys are saved to, and loaded from, files using the special format JKS (Java Key Store), also used by the API functions.

The PEM format, used by OpenSSL, is not supported by **keytool**. And the **openssl** command can’t deal with the JKS format either. Yet, there’s a format supported by both commands, it’s DER (Distinguished Encoding Rules), so by using it becomes possible copying a certificate from an OpenSSL based application to a Java based application.

2. Implementing a generic TLS client to test servers

In laboratory classes, we are going to use the OpenSSL C API on Linux systems, only.

For a first contact with the API, and a better understanding about the TLS/SSL handshake and public key certificates, we will implement a very simple TLS client to test TLS servers. It will simply create a TCP connection with a server, then secure the TCP connection with SSL/TLS, and then close the connection. Beyond the SSL/TLS handshake, no data is actually exchanged.

2.1. The source code (check_server_TLS.c)

We already know it all starts by creating a standard TCP connection, so the first part of the code is a standard TCP client, it receives the server’s DNS name on the command line, followed by an option port number, if omitted, it’s by default 443 (HTTPS).

Only once the TCP connection is established, then SSL/TLS starts operating, first a SSL context is prepared, including establishing the trusted certificates. Given the objective, we don’t want to impose restrictions regarding SSL/TLS versions and cypher suites.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/ssl.h>
#include <openssl/x509.h>
#include <openssl/x509_vfy.h>

#define BUF_SIZE 500
#define DEFAULT_SERVER_PORT "443"
```

```
int main(int argc, char **argv) {
       int err, sock;
       char line[BUF_SIZE];
       struct addrinfo  req, *list;
       char *serverPort=DEFAULT_SERVER_PORT;

       if(argc<2) {
              puts("HTTPS server's DNS name is required as argument");
              exit(1);
              }
       if(argc==3) serverPort=argv[2];
       bzero((char *)&req,sizeof(req));
       req.ai_family = AF_UNSPEC;
       req.ai_socktype = SOCK_STREAM;
       err=getaddrinfo(argv[1], serverPort , &req, &list);
       if(err) {
              printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1); }

       sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
       if(sock==-1) {
              perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

       if(connect(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
              perror("Failed connect"); freeaddrinfo(list); close(sock); exit(1);}

       puts("----------------------\nConnected (TCP)");

       const SSL_METHOD *method=SSLv23_method();
       SSL_CTX *ctx = SSL_CTX_new(method);

       // ABORT ON HANDSHAKE IF CERTIFICATE UNTRUSTED
       // SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,NULL);

       // LOAD TRUSTED ISSUERS CERTIFICATES LIST
       SSL_CTX_load_verify_locations(ctx,NULL,"/etc/ssl/certs");

       // Require HIGH, standing for 128-bits or above keys.
       // Disable some less safe cyphers
       // SSL_CTX_set_cipher_list(ctx,"HIGH:!aNULL:!kRSA:!PSK:!SRP:!MD5:!RC4");

       // SOME SERVERS MAY NOT SUPPORT TLS1.2, and thus the handshake would fail
       // To support lower versions, comment this
       // SSL_CTX_set_min_proto_version(ctx,TLS1_2_VERSION);

       SSL *sslConn = SSL_new(ctx);

       SSL_set_fd(sslConn, sock);
       if(SSL_connect(sslConn)!=1) {
              puts("SSL/TLS handshake error");
              SSL_free(sslConn); close(sock); exit(1);
       }
       puts("SSL/TLS handshake successful.");

       printf("Selected TLS version: %s\nSelected cypher suite: %s\n-----------------------\n",
                     SSL_get_cipher_version(sslConn),SSL_get_cipher(sslConn));

       X509* cert=SSL_get_peer_certificate(sslConn);
       X509_free(cert);

       if(cert==NULL) {
              puts("Sorry: no certificate was provided by the server, aborting.");
              SSL_free(sslConn); close(sock); exit(1);
       }

       X509_NAME_oneline(X509_get_subject_name(cert),line, BUF_SIZE);
       printf("Server's certificate subject: %s\n", line);

       char *cn=strstr(line,"/CN=");
       if(cn==NULL) {
```

```
                puts("SECURITY WARNING: certificate CN attribute not found");
        }
        else {
                cn+=4;  // remove any additional attribute
                char *nextAttr=strstr(cn,"/"); if(nextAttr) *nextAttr=0;
                if(strcmp(cn,argv[1])) {
                        printf("SECURITY WARNING: the certificate CN attribute (%s) doesn't match
the server's DNS name (%s).\n", cn, argv[1]);
                }
                else {
                        printf("OK:  the  server's  DNS  name  (%s)  matches  the  certificate's  CN
attribute.\n", argv[1]);
                }
        }

        X509_NAME_oneline(X509_get_issuer_name(cert),line,BUF_SIZE);
        printf("\nServer's certificate issuer: %s\n---------------------------\n", line);

        long result=SSL_get_verify_result(sslConn);
        if(result!=X509_V_OK) {
                puts("SECURITY WARNING: untrusted server certificate");
                printf("The certificate's problem is: %s\n",
                        X509_verify_cert_error_string(result));
        }
        else {
                puts("OK: The certificate is trusted.");
        }
        puts("------------------------");


        SSL_free(sslConn);
        close(sock);
        exit(0);
        }
```

The certificate received from the server, is checked by the API, but for now, that checking doesn´t include the DNS name (it’s already available on some API versions). So, the DNS name checking is very modestly performed by the application. Later, when testing, you may observe some certificates have wildcards in the CN attribute, that´s not covered by our implementation.

2.2. Compiling

When compiling programs using the OpenSSL API, the necessary libraries must be linked. In this case, it may be compiled with the following command line:

```
gcc -Wall check_server_TLS.c -o check_server_TLS -lssl -lcrypto
```

Of course, if you have download the **Makefile** file together with **check_server_TLS.c**, then the **make** command may be used to attain the same.

2.3. Testing SSL/TLS servers

Once compiled, we will now use the **./check_server_TLS** command (our program) to test public servers, the server DNS name must be provided on the command line, if no port number is specified, the program assumes port number 443 (HTTPS).

While testing, check which is the SSL/TLS version and cypher suite resulting from the TLS handshake, and also the received certificate, playing special attention to the CN attribute.

Testing suggestions:

- `./check_server_TLS www.dei.isep.ipp.pt`
- `./check_server_TLS rede.dei.isep.ipp.pt`
- `./check_server_TLS www.isep.ipp.pt`
- `./check_server_TLS portal.isep.ipp.pt`
- `./check_server_TLS moodle.isep.ipp.pt`
- `./check_server_TLS www.fccn.pt`
- `./check_server_TLS www.iana.com`
- `./check_server_TLS www.google.com`
- `./check_server_TLS www.facebook.com`
- `./check_server_TLS vsrv0s.dei.isep.ipp.pt 636`

All test suggestions are HTTPS servers, except for the last one, that’s the DEI LDAP server, in this case LDAPS (port number 636, instead of 389 that would be used for standard LDAP access, without SSL/TLS).

You may test whatever other servers come to your mind.

3. Securing network applications with SSL/TLS

One major advantage of SSL/TLS is that it can be easily added to any unsecure TCP based application protocol, without any change to the original protocol, and without completely redesigning the original applications.

We are going to demonstrate that by transforming two prior examples addressed in past laboratory lessons into secure versions. For that, two applications have been selected:

- The first TCP client/server we have implemented (tcp_cli_sum/tcp_srv_sum).
- The HTTP server we have implemented (http_srv_ajax_voting).

In each case a different scenario will be assumed, both using public key certificates-based authentication.

These two scenarios of public key, certificates-based authentication, have already been addressed on the previous laboratory lesson.

The use of SSL/TLS in Java is out of our scope, so we will focus on the C versions only.

4. Secure version of tcp_cli_sum/tcp_srv_sum

For this client/server applications, we want a closed scenario, where there's a known server instance and several known client instances, and where mutual authentication is a requisite between all applications.

Every application's instance (server and clients) will have each a unique self-signed certificate.

All client instances will trust the server's certificate only. The server will trust all clients' certificates that should be allowed access.

In the provided solution, beyond the server's certificate, certificates for four clients are generated (client1, client2, client3, and client4), yet the server will trust the certificates for client1, client2, and client3, but not for client4.

The included BASH script **make_certs**, may be called by running **make certs**, it will use the **openssl** command line utility to create five self-signed certificates, one for the server, and four more, one for each client.

The created files have the **.pem** extension for certificates and the **.key** extension for matching private keys:

```
client1.pem client1.key
client2.pem client2.key
client3.pem client3.key
client4.pem client4.key
server.pem server.key
```

The script will also create the **authentic-clients.pem** file containing client1, client2, and client3 certificates, but not the client4 certificate.

We are creating a closed environment where three clients will be allowed, each with its own certificate, though probably for most cases, a single client's certificate could do the trick. We can have several client instances using the same certificate at the same time.

4.1. The server - tcp_srv_sum_TLS.c

The main additions and changes to the original **tcp_src_sum.c** source file are highlighted in bold/blue:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

```
#include <openssl/crypto.h>
#include <openssl/bio.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define BUF_SIZE 300
#define SERVER_PORT "9999"

#define SERVER_SSL_CERT_FILE "server.pem"
#define SERVER_SSL_KEY_FILE "server.key"
#define AUTH_CLIENTS_SSL_CERTS_FILE "authentic-clients.pem"

int main(void) {
        struct sockaddr_storage from;
        int err, newSock, sock;
        unsigned int adl;
        unsigned long i, f, n, num, sum;
        unsigned char bt;
        char cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
        struct addrinfo  req, *list;

        bzero((char *)&req,sizeof(req));
        // requesting a IPv6 local address will allow both IPv4 and IPv6 clients to use it
        req.ai_family = AF_INET6;
        req.ai_socktype = SOCK_STREAM;
        req.ai_flags = AI_PASSIVE;      // local address

        err=getaddrinfo(NULL, SERVER_PORT , &req, &list);
        if(err) {
                printf("Failed to get local address, error: %s\n",gai_strerror(err)); exit(1);
}

        sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
        if(sock==-1) {
                perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

        if(bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
                perror("Bind failed");close(sock);freeaddrinfo(list);exit(1);}

        freeaddrinfo(list);
        listen(sock,SOMAXCONN);

        const SSL_METHOD *method;
        SSL_CTX *ctx;

        method = SSLv23_server_method();
        ctx = SSL_CTX_new(method);

        // Load the server's certificate and key
        SSL_CTX_use_certificate_file(ctx, SERVER_SSL_CERT_FILE, SSL_FILETYPE_PEM);
        SSL_CTX_use_PrivateKey_file(ctx, SERVER_SSL_KEY_FILE, SSL_FILETYPE_PEM);
        if (!SSL_CTX_check_private_key(ctx)) {
                puts("Error loading server's certificate/key");
                close(sock);
                exit(1);
        }

        // THE CLIENTS' CERTIFICATES ARE TRUSTED
        SSL_CTX_load_verify_locations(ctx, AUTH_CLIENTS_SSL_CERTS_FILE, NULL);

        // Restrict TLS version and cypher suite
        SSL_CTX_set_min_proto_version(ctx,TLS1_2_VERSION);
        SSL_CTX_set_cipher_list(ctx, "HIGH:!aNULL:!kRSA:!PSK:!SRP:!MD5:!RC4");

        // Clients must provide a trusted certificate, the handshake will fail otherwise
        SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);

        puts("Accepting TCP connections (IPv6 or IPv4). Use CTRL+C to terminate the server");

        adl=sizeof(from);
        for(;;) {
                newSock=accept(sock,(struct sockaddr *)&from,&adl);
                if(!fork()) {
                        close(sock);
                        getnameinfo((struct sockaddr *)&from,adl,cliIPtext,BUF_SIZE,
                                        cliPortText,BUF_SIZE,NI_NUMERICHOST|NI_NUMERICSERV);
                        printf("New connection from node %s, port number %s\n",
                                       cliIPtext, cliPortText);
                        SSL *sslConn = SSL_new(ctx);
                        SSL_set_fd(sslConn, newSock);
                        if(SSL_accept(sslConn)!=1) {
                                puts("TLS handshake error: client not authorized");
                                SSL_free(sslConn);
                                close(newSock);
                                exit(1);
                                }
                        printf("TLS version: %s\nCypher suite: %s\n",
                                   SSL_get_cipher_version(sslConn),SSL_get_cipher(sslConn));
                        X509* cert=SSL_get_peer_certificate(sslConn);
                        X509_free(cert);
                        X509_NAME_oneline(X509_get_subject_name(cert),cliIPtext,BUF_SIZE);
                        printf("Client's certificate subject: %s\n",cliIPtext);

                        do {
                                sum=0;
                                do {
                                        num=0;f=1;
                                        for(i=0;i<4;i++) {
                                                SSL_read(sslConn,&bt,1); num=num+bt*f; f=256*f;
                                                }
                                        sum=sum+num;
                                        }
                                while(num);
                                n=sum;
                                for(i=0;i<4;i++) {
                                        bt=n%256; SSL_write(sslConn,&bt,1); n=n/256; }
                                }
                        while(sum);
                        SSL_free(sslConn);
                        close(newSock);
                        printf("Connection from node %s, port number %s closed\n",
                                              cliIPtext, cliPortText);
                        exit(0);
                        }
                close(newSock);
                }
        close(sock);
        }
```

The **SSL_CTX_load_verify_locations()** function settles the trusted clients to those using a certificate present in file **authentic-clients.pem**.

The function call:

SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);

It's most important, the **SSL_VERIFY_PEER** flag forces the server to include in the TLS Server Hello message the demand for a client's certificate, and it will then be checked if it's trusted, and otherwise the handshake fails. The **SSL_VERIFY_FAIL_IF_NO_PEER_CERT**, forces the handshake to fail if the client fails to provide a certificate.

As we can see, regarding data exchanges once the handshake has successfully ended, it's just a matter of replacing the original read() and write() functions with OpenSSL API equivalents: SSL_read() and SSL_write().

4.2. The client - tcp_cli_sum_TLS.c

Beyond the server's IP address, as with the original application, now a second argument is used on the command line. It's the base filename from where to load the client's public key certificate and corresponding private key, if not specified, then the client will not use a certificate.

The main additions and changes to the original **tcp_cli_sum.c** source file are highlighted in bold/blue:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <openssl/conf.h>
#include <openssl/x509.h>

#define SERVER_SSL_CERT_FILE "server.pem"

#define BUF_SIZE 60
#define SERVER_PORT "9999"

// read a string from stdin protecting buffer overflow
#define GETS(B,S) {fgets(B,S-2,stdin);B[strlen(B)-1]=0;}

int main(int argc, char **argv) {
        int err, sock;
        unsigned long f, i, n, num;
        unsigned char bt;
        char line[BUF_SIZE];
        struct addrinfo  req, *list;

        if(argc<2) {
            printf("\nUsage:\n\n%s SERVER-IP [CLIENT-NAME]\n\n",argv[0]);
            puts("SERVER-IP: Server's IPv4/IPv6 address or DNS name.");
            puts("CLIENT-NAME: base filename to load client's certificate and private key.");
            puts("             !!! If not provided, the client will not use a certificate.\n");
            exit(1);
            }

        bzero((char *)&req,sizeof(req));
        // let getaddrinfo set the family depending on the supplied server address
        req.ai_family = AF_UNSPEC;
        req.ai_socktype = SOCK_STREAM;
        err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
        if(err) {
                printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1);
}

        sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
        if(sock==-1) {
                perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

        if(connect(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
                perror("Failed connect"); freeaddrinfo(list); close(sock); exit(1);}
        puts("Connected");


        const SSL_METHOD *method=SSLv23_client_method();
        SSL_CTX *ctx = SSL_CTX_new(method);

        if(argc==3) {
           // Load client's certificate and key
           strcpy(line,argv[2]);strcat(line,".pem");
           SSL_CTX_use_certificate_file(ctx, line, SSL_FILETYPE_PEM);
           strcpy(line,argv[2]);strcat(line,".key");
           SSL_CTX_use_PrivateKey_file(ctx, line, SSL_FILETYPE_PEM);
           if (!SSL_CTX_check_private_key(ctx)) {
                puts("Error loading client's certificate/key");
                close(sock);
                exit(1);
                }
        }

        SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,NULL);

        // THE SERVER'S CERTIFICATE IS TRUSTED
        SSL_CTX_load_verify_locations(ctx,SERVER_SSL_CERT_FILE,NULL);

        // Restrict TLS version and cypher suites
        SSL_CTX_set_min_proto_version(ctx,TLS1_2_VERSION);
        SSL_CTX_set_cipher_list(ctx, "HIGH:!aNULL:!kRSA:!PSK:!SRP:!MD5:!RC4");

        SSL *sslConn = SSL_new(ctx);
        SSL_set_fd(sslConn, sock);
        if(SSL_connect(sslConn)!=1) {
                puts("TLS handshake error");
                SSL_free(sslConn);
                close(sock);
                exit(1);
        }
        printf("TLS version: %s\nCypher suite: %s\n",
                    SSL_get_cipher_version(sslConn),SSL_get_cipher(sslConn));

        if(SSL_get_verify_result(sslConn)!=X509_V_OK) {
                puts("Sorry: invalid server certificate");
                SSL_free(sslConn);
                close(sock);
                exit(1);
        }

        X509* cert=SSL_get_peer_certificate(sslConn);
        X509_free(cert);

        if(cert==NULL) {
                puts("Sorry: no certificate provided by the server");
                SSL_free(sslConn);
                close(sock);
                exit(1);
        }

        do {
                do {
                        printf("Enter a positive integer to SUM (zero to terminate): ");
                        GETS(line,BUF_SIZE);
                        while(sscanf(line,"%li",&num)!=1 || num<0) {
                                puts("Invalid number");
                                GETS(line,BUF_SIZE);
                                }
                        n=num;
                        for(i=0;i<4;i++) {
                                bt=n%256; SSL_write(sslConn,&bt,1); n=n/256; }
                        }
                while(num);
                num=0; f=1; for(i=0;i<4;i++) {SSL_read(sslConn,&bt,1); num=num+bt*f; f=f*256;}
                printf("SUM RESULT=%lu\n",num);
                }
        while(num);
        SSL_free(sslConn);
        close(sock);
        exit(0);
        }
```

4.3. Testing

The deployment scenario is shown in Figure 1.

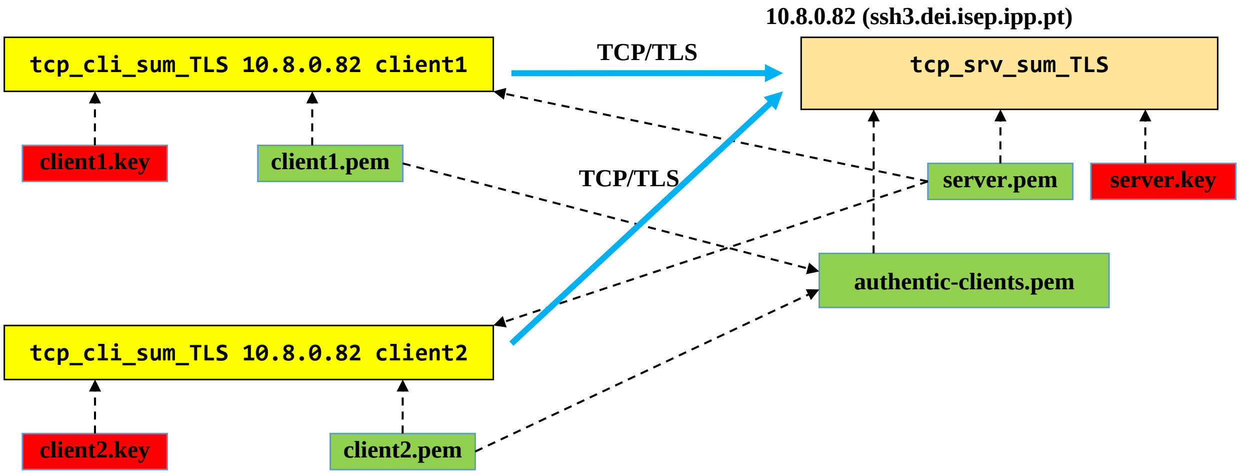

Figure 1 - Secure TCP client/server communications with public key certificates authentication

In Figure 1, the dashed lines represent certificate and private keys reading and sharing, notice that private keys (in red) are never shared. Certificates and private keys for two additional clients will be generated (**client3.pem/client3.key** and **client4.pem/client4.key**), however **client4.pem** will not be added to the **authentic-clients.pem**, thus it will not be trusted/accepted by the server.

Pick one specific Linux SSH server to run **tcp_srv_sum_TLS**, we will assume to be ssh3, it will be accessible by client through the DNS name labs-ssh3 (10.8.0.82).

Change the port number, to avoid collisions with other students running **tcp_srv_sum_TLS** in the same Linux SSH server, each should use a different port number. Change it both on the server and client source code.

In no specific order, build the applications and create all certificates and private key files:

```
make
make certs
```

Now you can start the server on ssh3:

```
./tcp_srv_sum_TLS
```

Pick a different Linux SSH server to run tcp_cli_sum_TLS and provide the required arguments:

```
./tcp_cli_sum_TLS labs-ssh3 client1
```

We are using the public key certificate belonging to client1, it should work. The application should work the same way as the original one.

Test again with the other public key certificates belonging to clients: client2, client3, and client4.

For client4 it will fail, that certificate is not trusted by the server.

Test the client with no certificate at all, it should fail as well. Our server demands a certificate (SSL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT).

You may test several other scenarios, for instance with several clients connected at the same time, it will work even if all clients are using the same certificate.

4.4. The real-world scenario

What we have tested was deployed in an experimental scenario, the real thing would something like:

Over the internet there are 3 hosts, named Host-A, Host-B, and Host-C. Each with its own administrator, respectively, Administrator-A, Administrator-B, and Administrator-C.

Administrator-A will run the server on Host-A, Administrator-B will run a client on Host-B, and Administrator-C will run a client on Host-C.

- Administrator-A would create a self-signed public key certificate to be used by the local server application, and send it to Administrator-B and Administrator-C.
- Administrator-B and Administrator-C would set the public key certificate received from Administrator-A as the only one trusted by their client applications.
- Administrator-B and Administrator-C would create each a self-signed public key certificates to be used by the local client application, and each would send it to Administrator-A.
- Administrator-A would add (if he decides to) the certificates received from Administrator-B, and from Administrator-C, to the list of trusted certificates for the local server applications.

Without sending any secret to other administrators, all applications are authenticated. Clients are sure they are talking with the correct server because the only certificate that they trust is the one belonging to the server (manually added by the local administrator). The server is sure is talking with the correct clients because the only certificates it trusts are the ones that belong to clients (manually added by the local administrator).

This is true as far as administrators trust each other, namely Administrator-B and Administrator-C trust Administrator-A, and Administrator-A trusts Administrator-B and Administrator-C.

5. Secure version of http_srv_ajax_voting (HTTPS instead of HTTP)

Due to the bigger complexity of this application's source code, several widespread modifications are required, especially regarding functions prototypes that were receiving the TCP connection to the client as a socket, and they must now be converted to receive a SSL connection instead.

Beyond this detail, changes are as before minimal. As before calls to read() and write() functions must be replaced with OpenSSL API equivalents: SSL_read() and SSL_write().

We will focus on the most significant changes within the main() function in the application file **https_srv_ajax_voting.c**, again with changes highlighted in bold/blue.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <signal.h>

#include <openssl/crypto.h>
#include <openssl/bio.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#include "http.h"

#define BASE_FOLDER "www"
```

```
#define SERVER_SSL_CERT_FILE "server.pem"
#define SERVER_SSL_KEY_FILE "server.key"

void processHttpRequest(int sock, int conSock, SSL *sslConn); // implemented ahead
void processGET(SSL *sslConn, char *requestLine);       // implemented ahead
void processPUT(SSL *sslConn, char *requestLine);       // implemented ahead

#define NUM_CANDIDATES 4
char *candidateName[] = { "Candidate A", "Candidate B", "Candidate C" , "Candidate D" };
int candidateVotes[NUM_CANDIDATES];
unsigned int httpAccessesCounter=0;

int main(int argc, char **argv) {
        struct sockaddr_storage from;
        int err, i, newSock, sock;
        socklen_t adl;
        struct addrinfo  req, *list;

        if(argc!=2) {puts("Oops, local TCP port number missing from command line"); exit(1);}
        bzero((char *)&req,sizeof(req));
        req.ai_family = AF_INET6;
        req.ai_socktype = SOCK_STREAM;
        req.ai_flags = AI_PASSIVE;      // local address
        err=getaddrinfo(NULL, argv[1] , &req, &list);
        if(err) {
                printf("Failed to get local address, error: %s\n",gai_strerror(err)); exit(1);
}

        sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
        if(sock==-1) {
                perror("Failed to open socket"); freeaddrinfo(list); exit(1);}
        if(bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
                perror("Bind failed");close(sock);freeaddrinfo(list);exit(1);}
        freeaddrinfo(list);
        listen(sock,SOMAXCONN);

        const SSL_METHOD *method = SSLv23_server_method();
        SSL_CTX *ctx = SSL_CTX_new(method);

        // Load the server's certificate and key
        SSL_CTX_use_certificate_file(ctx, SERVER_SSL_CERT_FILE, SSL_FILETYPE_PEM);
        SSL_CTX_use_PrivateKey_file(ctx, SERVER_SSL_KEY_FILE, SSL_FILETYPE_PEM);
        if (!SSL_CTX_check_private_key(ctx)) {
                puts("Error loading server's certificate/key");
                close(sock);
                exit(1);
        }

        SSL_CTX_set_cipher_list(ctx, "HIGH:!aNULL:!kRSA:!PSK:!SRP:!MD5:!RC4");

        adl=sizeof(from);
        for(i=0; i<NUM_CANDIDATES; i++) candidateVotes[i]=0;
        signal(SIGCHLD, SIG_IGN); // AVOID LEAVING TERMINATED CHILD PROCESSES AS ZOMBIES
        while(1) {
                newSock=accept(sock,(struct sockaddr *)&from,&adl);
                httpAccessesCounter++;
                SSL *sslConn = SSL_new(ctx);
                SSL_set_fd(sslConn, newSock);
                if(SSL_accept(sslConn)!=1) {
                        puts("TLS handshake error.");
                        SSL_free(sslConn);
                        close(newSock);
                        }
                else {
                        processHttpRequest(sock,newSock,sslConn);
                  }
                }
        close(sock);
        return(0);
        }
```

Unlike the previous server, this does not demand for a client's certificate.

5.1. Testing

As before, a small BASH script is provided to create a server's self-signed certificate. **And, self-signed must be emphasized now**. This means when we later test the server with a standard Web Browser, the issuer will not be trusted.

If we were to make such a server publicly available, we would have to attain a public key certificate matching the server's DNS name, issued by a worldwide trusted CA.

Pick one specific Linux SSH server to run **https_srv_ajax_voting**, we will assume to be ssh2, it will be accessible by client through the DNS name labs-ssh2.

Select one Linux SSH server to run the application, we will assume

Compile the application and create the self-signed certificate for the server, in no particular sequence:

```
make
make cert
```

As usual, because these SSH servers are shared among students, establish a port number to be used (**PORT-NUMBER**), different from other students, and start the server by running:

```
./ https_srv_ajax_voting PORT-NUMBER
```

On a standard Web Browser, type the URL:

```
https://labs-ssh2.dei.isep.ipp.pt:PORT-NUMBER
```

It must be started by **https://** and not http://, otherwise the browser will not use TLS.

Of course, the client host, running the browser, must be connected to a DEI private network.

This application is using AJAX, notice that JavaScript requests are made through HTTPS as well. This is because requests are relative to the main HTML page's **origin** (**https://labs-ssh2.dei.isep.ipp.pt:PORT-NUMBER**), itself HTTPS.