Katedra Informatyki

Wydział Informatyki i Telekomunikacji
Politechnika Krakowska

Programowanie Usług Sieciowych

mgr inż. Michał Niedźwiecki

Bezpieczna komunikacja w OpenSSL

Laboratorium: 10,

system operacyjny: Linux

Kraków, 2022

Spis treści

1. Wiadomości wstępne

Pierwsza część niniejszej instrukcji zawiera podstawowe wiadomości teoretyczne dotyczące protokołów SSL/TLS oraz programowania aplikacji wykorzystujących API OpenSSL. Poznanie tych wiadomości umożliwi prawidłowe zrealizowanie praktycznej części laboratorium.

1.1. Tematyka laboratorium

Tematyką laboratorium jest programowanie aplikacji klient-serwer wykorzystujących protokół SSL oraz jego następcę, protokół TLS. Protokoły SSL/TLS umożliwiają tworzenie bezpiecznych połączeń do wymiany danych między dwoma stacjami sieciowymi. Niniejsza instrukcja poświęcona jest popularnej, darmowej implementacji protokołów SSL/TLS – bibliotece OpenSSL.

OpenSSL składa się z trzech zasadniczych części:

- biblioteki kryptograficznej (libcrypto),
- biblioteki SSL (libssl),
- programu openssl.

Biblioteka libcrypto udostępnia API dla algorytmów kryptograficznych i zapewnia obsługę PKI (ang. Public Key Infrastructure).

Biblioteka libssl wykorzystuje funkcjonalność libcrypto do implementacji protokołów SSLv2, SSLv3 oraz TLSv1.

Program openssl pozwala na wykorzystanie biblioteki kryptograficznej z poziomu linii poleceń lub skryptów powłoki. Umożliwia on m.in.: tworzenie certyfikatów X.509, obliczanie kryptograficznych skrótów wiadomości, szyfrowanie i deszyfrowanie plików.

1.2. Zagadnienia do przygotowania

Przed przystąpieniem do realizacji laboratorium należy zapoznać się z zagadnieniami dotyczącymi gniazd sieciowych i protokołów SSL/TLS: [1 - 5]

- reprezentacja adresów IP i porządek bajtów
- tworzenie gniazd klienta i serwera dla protokołu TCP
- operacje I/O na gniazdach
- nawiązywanie połączenia SSL/TLS
- zestawy szyfrów wykorzystywane przez SSL/TLS
- podstawowe różnice między SSLv2, SSLv3, a TLSv1
- kompatybilność SSLv3 i TLSv1
- certyfikaty i nazwy wyróżniające

Ponadto, wymagana jest: [6 - 9]

- umiejętność generowania kluczy RSA, tworzenia żądań certyfikacyjnych i wystawiania certyfikatów za pomocą programu opensll
- znajomość podstawowych funkcji API OpenSSL
- umiejętność obsługi sniffera tshark

Literatura:

[1] W.R. Stevens, „Programowanie Usług Sieciowych”, „API: gniazda i XTI”

[2] S. A. Thomas, „SSL and TLS essentials: Securing the Web”, Wiley.

[3] IETF, RFC 2246, „The TLS Protocol Version 1.0”

[4] IETF, RFC 4346, „The Transport Layer Security (TLS) Protocol Version 1.1”

[5] IETF, RFC 5246, „The Transport Layer Security (TLS) Protocol Version 1.2”

[6] P. Chandra, M. Messier, J. Viega, „Network Security with OpenSSL”, O’Reilly

[7] MAN (3ssl), „ssl”, „dh”

[8] MAN (1ssl), „genrsa”, „req”, „ca”, „x509”, „ciphers”, „c_rehash”

[9] MAN (1), „tshark”

1.3. Opis laboratorium

1.3.1. Gniazda TCP

Z uwagi na fakt, że specyfikacja SSL/TLS wymaga zastosowania niezawodnego protokołu warstwy transportowej, niniejszy podrozdział poświęcony jest tworzeniu i wykorzystywaniu gniazd TCP.

Gniazda (ang. sockets) są najbardziej uniwersalnym sposobem komunikacji spośród mechanizmów IPC (ang. Inter Process Communication - komunikacja międzyprocesowa). Jeśli dwa procesy mają się między sobą komunikować, każdy z nich tworzy po swojej stronie jedno gniazdo. Parę takich gniazd (końcówek kanału komunikacyjnego) można określić mianem asocjacji. Gniazda reprezentują najczęściej dwukierunkowy punkt końcowy połączenia – dwukierunkowość oznacza możliwość wysyłania i przyjmowania danych.

TCP jest protokołem połączeniowym – przed wymianą danych, konieczne jest ustanowienie połączenia. Połączenie TCP stanowi abstrakcyjny, dwukierunkowy kanał, którego końce (gniazda) są reprezentowane przez adres IP i numer portu. Proces ustanawiania połączenia nosi nazwę three-way handshake i rozpoczyna się, gdy klient wysyła do serwera pakiet TCP z ustawionym bitem SYN. Po stronie serwera, specjalny typ gniazda nasłuchuje na przychodzące połączenia. Dla każdego zaakceptowanego połączenia

tworzone jest tzw. gniazdo połączone (ang. connected socket). Jego zadaniem jest obsługa operacji I/O związanych z danym połączeniem TCP. Proszę zwrócić uwagę na fakt, że serwer wykorzystuje zarówno gniazdo nasłuchujące (ang. listening socket), jak i połączone.

W systemach operacyjnych Linux gniazda są tworzone za pomocą funkcji socket(). W celu utworzenia gniazda dla protokołu TCP, należy określić:

- domenę komunikacyjną (rodzinę protokołów), w obrębie której odbywa się komunikacja: PF_INET dla IPv4 lub PF_INET6 dla IPv6,
- typ gniazda określający sposób transmisji: SOCK_STREAM – niezawodna, dwukierunkowa transmisja za pomocą strumienia bajtów,
- protokół (0 oznacza domyślny protokół dla danej domeny komunikacyjnej i typu gniazda).

Funkcja socket() zwraca deskryptor gniazda lub wartość -1 w przypadku błędu:

```
int sockfd = socket(PF_INET, SOCK_STREAM, 0);

if (sockfd == -1) {
    perror("socket()");
    exit(EXIT_FAILURE);
}
```

Utworzone za pomocą funkcji socket() gniazdo nie jest ani nasłuchujące, ani połączone. Aby przekształcić gniazdo w gniazdo połączone należy wywołać funkcję connect(). Funkcja connect() ustanawia połączenie klienta z serwerem poprzez inicjację

trójfazowego połączenia (ang. three-way handshake).

```
#include <sys/socket.h>

int connect(int sockfd,
    const struct sockaddr *servaddr,
    socklen_t ddrlen);
```

Parametrami funkcji connect() są:

- deskryptor gniazda,
- wskaźnik do gniazdowej struktury adresowej,
- rozmiar struktury adresowej.

Poniżej przedstawiony jest przykład nawiązania połączenia TCP z serwerem nasłuchującym pod adresem 127.0.0.1, na porcie 80.

```
/* Struktura adresowa dla IPv4: */
struct sockaddr_in server_addr;

/* Konwencją jest wyzerowanie struktury: */
memset(&server_addr, 0, sizeof(server_addr));

/* Rodzina protokołów - IPv4: */
server_addr.sin_family = AF_INET;

/* Adres zdalny (serwera) - 127.0.0.1: */
inet_pton(AF_INET, “127.0.0.1”, &server_addr.sin_addr);

/* Port zdalny - 80: */
server_addr.sin_port = htons(80);

/* Nawiązanie połączenia TCP: */
connect(sockfd,
    (const struct sockaddr*)&server_addr,
    sizeof(server_addr));
```

Istnieje kilka różnych struktur adresowych, ale w przykładzie została wykorzystana struktura dla protokołu IPv4 (sockaddr_in). Funkcja connect() nie przyjmuje jednak wskaźnika na sockaddr_in, a wskaźnik na ogólną strukturę adresową - sockaddr. Wynika to z faktu, że funkcje operujące na gniazdach powstały przed opracowaniem standardu ANSI C (typ void nie istniał). Dzięki zastosowaniu wskaźnika do struktury sockaddr, funkcje operujące na gniazdach posiadają jednolity interfejs umożliwiający przyjmowanie struktur adresowych różnych protokołów. Wskaźniki na struktury adresowe konkretnych protokołów (np. IPv4 i IPv6) można bezpiecznie rzutować na wskaźnik do struktury sockaddr. Rozmiar właściwej struktury jest przekazywany jako ostatni parametr funkcji connect().

Poniżej przedstawiona jest struktura sockaddr_in przechowująca informacje adresowe dla protokołu IPv4:

```
#include <netinet/in.h>

struct sockaddr_in {
    short int sin_family; /* AF_INET */
    unsigned short sin_port; /* Numer portu */
    struct in_addr sin_addr; /* Adres IP */
    unsigned char sin_zero[8] /* Wyzerowane */
};
```

Liczba pól struktury może różnić się w zależności od systemu operacyjnego, ale zawsze występują 3 pola:

- sin_family - domena adresowa (AF_INET dla IPv4),
- sin_port - numer portu przechowywany w porządku sieciowym (big endian),
- sin_addr - adres IP w porządku sieciowym; struktura in_addr przedstawiona jest poniżej:

```
#include <netinet/in.h>

struct in_addr {
    unsigned int s_addr;
};
```

Konwencją jest wyzerowanie całej struktury sockaddr_in przed jej użyciem, np. za pomocą funkcji memset().

Po ustanowieniu połączenia TCP za pomocą funkcji connect(), klient jest gotowy do wymiany danych. W przypadku serwera, aby osiągnąć gotowość do przeprowadzania operacji I/O, należy wywołać cztery funkcje:

- socket(),
- bind(),
- listen(),
- accept().

Funkcja bind() przypisuje gniazdu informacje adresowe (dla protokołu IP jest to kombinacja adresu IP oraz 16 bitowego numeru portu):

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

Parametr	Opis
sockfd	deskryptor gniazda
*addr	wskaźnik do struktury adresowej
addrlen	rozmiar struktury adresowej

Struktura adresowa może określać:

1. konkretny numer portu - najczęściej w przypadku serwera,
2. numer portu równy 0 – system operacyjny wybierze port efemeryczny (z dostępnych portów),
3. konkretny adres IP,
4. adres nieokreślony (ang. wildcard address) – w przypadku IPv4 jest to INADDR_ANY. Serwer będzie nasłuchiwał na wszystkich dostępnych interfejsach sieciowych.

Podobnie jak w przypadku funkcji connect(), wskaźnik do struktury adresowej konkretnego protokołu (np.: sockaddr_in dla IPv4) należy rzutować na wskaźnik do ogólnej struktury sockaddr.

Funkcja listen() jest wywoływana przez serwer w celu przekształcenia gniazda niepołączonego w gniazdo nasłuchujące, które będzie używane do akceptacji przychodzących połączeń TCP.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Parametr	Opis
sockfd	deskryptor gniazda
backlog	długość kolejki – ogranicza liczbę połączeń oczekujących dla danego gniazda; podana wartość jest modyfikowana przez system operacyjny

Ostatnia z wymienionych funkcji – accept() - pobiera połączenie TCP z kolejki połączeń oczekujących na zaakceptowanie, tworzy nowe gniazdo do celów komunikacji z klientem i zwraca jego deskryptor. Nowe gniazdo (połączone) może zostać wykorzystane do przeprowadzenia operacji I/O.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);
```

Parametr	Opis
sockfd	deskryptor gniazda
*cliaddr	wskaźnik do struktury adresowej wypełnianej przez funkcję accept() adresem gniazda klienta
*addrlen	określa rozmiar struktury wskazanej przez cliaddr

Wywołania systemowe write(), send() są wykorzystywane do wysyłania danych za pomocą gniazda połączonego:

```
ssize_t write(int sockfd, const void *buf, size_t len);
ssize_t send(int sockfd, const void *buf, size_t len, int
flags);
```

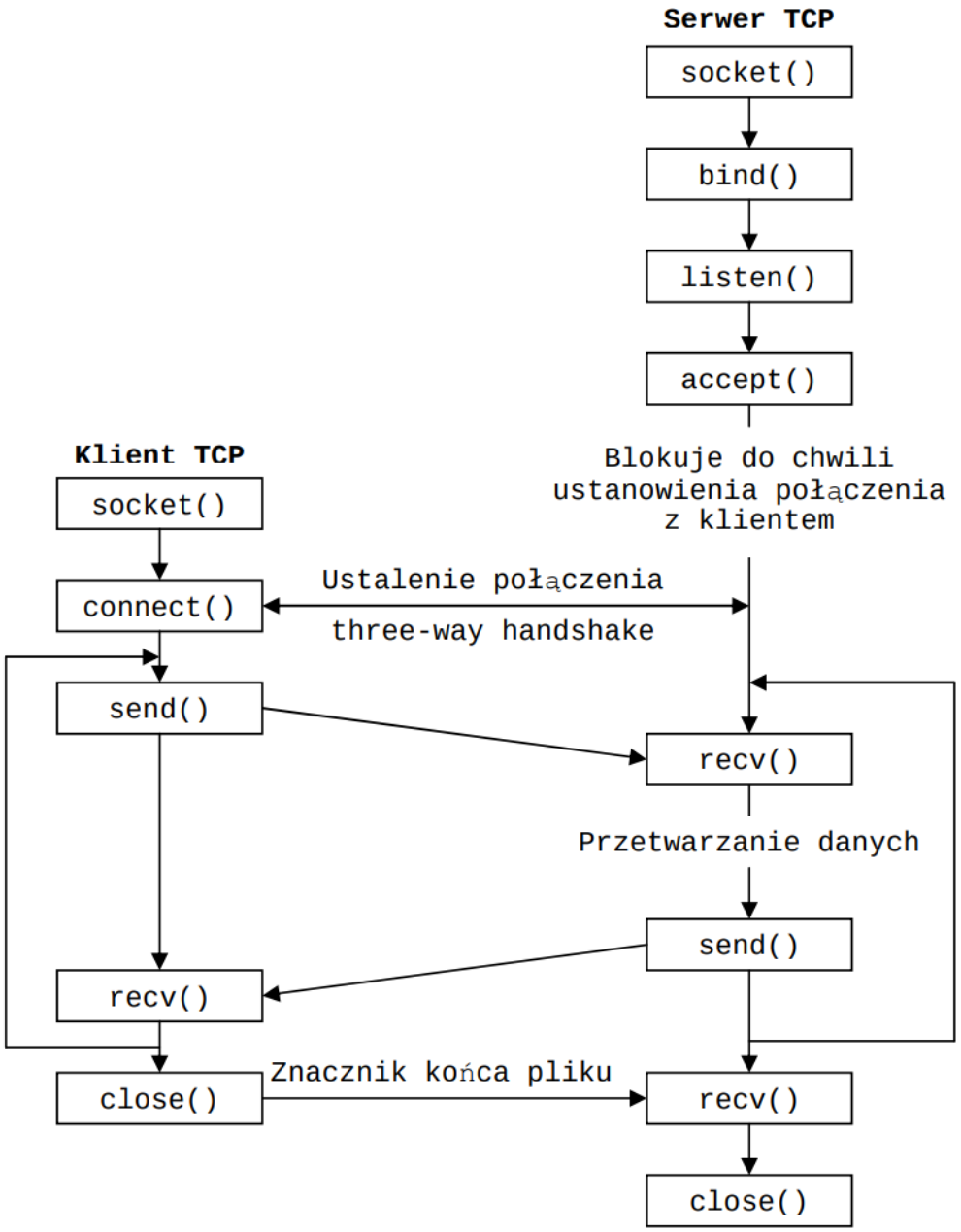
Parametr	Opis
sockfd	deskryptor gniazda

buf	bufor zawierający dane do wysłania
len	rozmiar danych do wysłania w buforze (w bajtach)
flagi	opcjonalne flagi

Analogicznie, wywołania systemowe read(), recv() można wykorzystać do odbierania danych z gniazda połączonego:

```
ssize_t read(int sockfd, void *buf, size_t len);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Rys. 1 przedstawia przebieg typowej komunikacji między klientem i serwerem TCP.



Rys. 1. Przebieg typowej komunikacji między klientem i serwerem TCP.

1.3.2. Protokoły SSL/TLS

Protokół SSL (ang. Secure Sockets Layer) został opracowany przez firmę Netscape w celu zabezpieczania komunikacji w sieci Internet, przede wszystkim dla potrzeb handlu elektronicznego. Prace nad pierwszą wersją protokołu SSL zostały zakończone w 1994 roku, ale specyfikacja SSLv1 nigdy nie została opublikowana.

Publikacji doczekała się za to wersja druga (1995r.). Pierwszym produktem zapewniającym wsparcie dla SSLv2 była przeglądarka Netscape Navigator. Aby wyeliminować pewne wady występujące w protokole SSLv2, firma Microsoft rozpoczęła prace nad własną specyfikacją protokołu – PCT (ang. Private Communication Technology). Część pomysłów wykorzystanych do opracowania SSLv3 (1995r.) zostało zaczerpnięte wprost z PCT. Następujące słabości protokołu SSLv2 były przyczyną opracowania SSLv3:

- W SSLv2 zakończenie połączenia TCP (wysłanie segmentu TCP z ustawioną flagą FIN) sygnalizuje koniec połączenia SSL. Atakujący może wykorzystać ten fakt do przeprowadzenia ataku DoS (truncation attack).
- SSLv2 umożliwia przeprowadzenie ataku typu downgrade; atakujący może wymusić zastosowanie zestawu słabych algorytmów kryptograficznych.
- SSLv2 używa tylko algorytmu MD5 do obliczenia kodu uwierzytelniającego wiadomości. Ponadto, te same klucze są wykorzystywane do szyfrowania i uwierzytelniania wiadomości.

W 1996 roku organizacja IETF powołała grupę roboczą o nazwie TLS (ang. Transport Layer Security). Jej głównym celem była standaryzacja protokołu SSL i niedopuszczenie do sytuacji, w której mogłyby funkcjonować wersje protokołu pochodzące od różnych firm. Wynikiem pracy grupy roboczej była publikacja protokołu TLSv1 (RFC 2246, 1999r.). W stosunku do SSLv3, protokół TLSv1:

- Definiuje więcej typów wiadomości dla protokołu Alert (wiadomości tego protokołu pełnią funkcje kontrolne i sygnalizujące).
- Zmienia sposób obliczania MAC. SSL wykorzystuje do tego celu własny algorytm, TLS bazuje na HMAC (RFC 2104).

- Zmienia sposób generowania kluczy kryptograficznych. TLS wykorzystuje PSF (ang. Pseudorandom Function) – procedurę bazującą na HMAC.
- Upraszcza format wiadomości CertificateVerify i Finished.

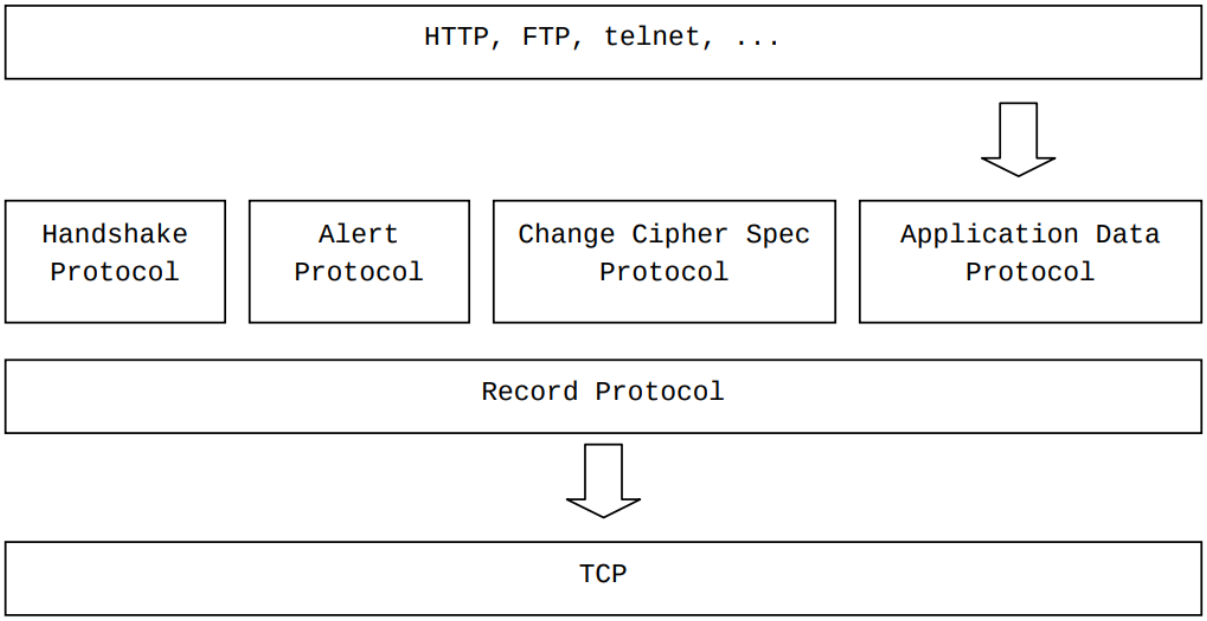
Powyższe modyfikacje nie są tak istotne dla bezpieczeństwa komunikacji, jak zmiany wprowadzone między wersjami SSLv2 i SSLv3.

Aktualną wersją protokołu jest TLSv1.2. Definiuje ona nowe zestawy algorytmów kryptograficznych, wprowadza mechanizm rozszerzeń (RFC 3546) i modyfikuje niektóre procedury i praktyki kryptograficzne stosowane w poprzednich wersjach.

Protokoły SSL/TLS, niezależnie od wersji, wymagają niezawodnego protokołu warstwy transportowej - w praktyce jest to TCP. Istnieje jednak wariant protokołu, który wykorzystuje datagramy UDP. Nosi on nazwę DTLS (ang. Datagram Transport Layer Security, RFC 4347).

Głównym celem protokołu TLS (SSL) jest zapewnienie poufności i integralności danych przesyłanych między komunikującymi się aplikacjami. Protokół składa się z dwóch warstw: TLS Record Protocol oraz TLS Handshaking Protocols. Bezpośrednio nad warstwą transportową (protokołem TCP) znajduje się TLS Record Protocol. Z usług TLS Record Protocol korzystają cztery protokoły (rys. 2):

- Handshaking Protocols: Handshake Protocol, Alert Protocol, Change Cipher Spec Protocol
- Application Data Protocol



Rys.2. Warstwy protokołu TLS.

Funkcje poszczególnych protokołów przedstawia skrótowo poniższa tabela:

Protokół	Funkcja
Change Cipher Spec	Protokół definiuje tylko jedną wiadomość. Jej celem jest poinformowanie zdalnej stacji o zmianie algorytmów kryptograficznych wykorzystywanych do dalszej komunikacji.
Alert	Protokół Alert pełni funkcję kontrolna i sygnalizująca. Wiadomości Alert informują zdalną stację o błędach komunikacji, możliwym złamaniu bezpieczeństwa, itp. Protokół Alert umożliwia bezpieczne zakończenie połączenia SSL/TLS.
Handshake	Odpowiedzialny za ustanowienie bezpiecznego połączenia (negocjację algorytmów kryptograficznych i pre-master

	secret, uwierzytelnianie).
Application Data	Właściwe dane (np. HTTP) transmitowane w formie zaszyfrowanej między dwoma stronami.
Record	Enkapsulacja i świadczenie usług dla protokołów: Change Cipher Spec, Alert, Handshake i Application Data.

TLS Record Protocol posiada dwie podstawowe cechy:

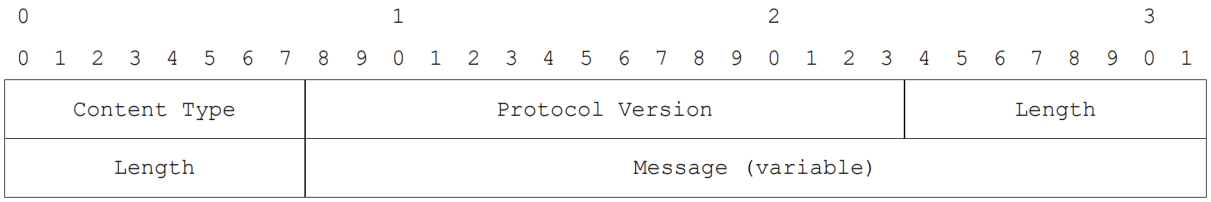
- Umożliwia tworzenie połączeń, które zapewniają poufność danych. W tym celu wykorzystuje szyfry symetryczne (AES, RC4, itp.). Unikatowe klucze są generowane dla każdego połączenia na podstawie sekretnych danych wynegocjowanych przez TLS Handshake Protocol.
- Wiadomości są transportowane z kodem uwierzytelniającym (MAC), na podstawie którego weryfikowana jest ich integralność.

TLS Record Protocol enkapsuluje protokoły warstw wyższych, w tym TLS Handshaking Protocols. Podczas wysyłania i odbierania danych, TLS Record Protocol jest odpowiedzialny za:

- fragmentację danych, kompresję, obliczanie MAC, szyfrowanie i transmisję,
- deszyfrowanie, weryfikację MAC, dekompresję, odtworzenie wiadomości z fragmentów i przekazanie wiadomości do protokołu warstwy wyższej.

Nagłówek protokołu Record Protocol jest transmitowany w formie niezaszyfrowanej.

Jego budowę przedstawia rys.3.



Rys.3. Budowa nagłówka Record Protocol.

Znaczenie poszczególnych pól:

- Content Type (1 bajt) – identyfikator przenoszonego protokołu:

Identyfikator	Protokół
20	*Change Cipher Spec*
21	*Alert*
22	*Handshake*
23	*Application Data*

- Protocol Version (2 bajty) - wersja protokołu:

Starszy bajt	Młodszy bajt	Wersja
3	*0*	*SSLv3*
3	*1*	*TLSv1.0*
3	*2*	*TLSv1.1*
3	*3*	*TLSv1.2*

- Length (2 bajty) – długość transportowanej wiadomości w bajtach.

Nagłówek Record Protocol wraz z wiadomością, którą przenosi nazywa się rekordem. Jeden segment TCP może przenosić wiele rekordów Record Protocol.

TLS Handshaking Protocols pozwalają przeprowadzić proces uwierzytelniania serwera i klienta oraz wynegocjować algorytmy kryptograficzne i klucze, stosowane w celu zapewnienia bezpiecznej komunikacji. Proces uwierzytelniania i negocjacji parametrów kryptograficznych opiera się na następujących założeniach:

- Tożsamość zdalnej stacji może zostać zweryfikowana za pomocą kryptografii klucza publicznego (RSA, DSA, itp.). Uwierzytelnianie jest procesem opcjonalnym, ale z reguły jest przeprowadzane dla co najmniej jednej ze stron.
- Negocjacja tajnych danych (tzw. pre-master secret), na podstawie których zostaną wygenerowane klucze jest bezpieczna. Osoby niepowołane nie mogą przechwycić tajnych danych.
- Proces negocjacji parametrów połączenia jest niezawodny. Atakujący nie może zakłócić procesu negocjacji (np. przez modyfikację danych) bez zostania wykrytym.

1.3.3. Ustanawianie połączenia TLS

Następujące etapy opisują proces ustanawiania połączenia TLS:

1. Klient wysyła do serwera wiadomość ClientHello. Wiadomość zawiera:
    - Wersję protokołu SSL/TLS jaką klient jest w stanie obsłużyć. Serwer może uznać, że klient obsługuje również wersje starsze od przedstawionej.
    - 32-bajtową, losową liczbę (w tym 4 bajty stanowi aktualna data i czas). Liczba posłuży do wygenerowania materiału kryptograficznego – tzw. master secret. Data i czas przedstawiona jest w

formie liczby sekund od 1 stycznia 1970 roku UTC.

- Identyfikator sesji (opcjonalnie, 32 bajty). Identyfikator jest obecny, jeżeli klient chce wznowić sesję (ponownie wykorzystać do komunikacji wynegocjowane wcześniej algorytmy). Brak identyfikatora oznacza, że klient żąda utworzenia nowej sesji.
- Listę zestawów algorytmów kryptograficznych w kolejności preferowanej przez klienta. Każdy zestaw definiuje algorytm wymiany klucza, algorytm wykorzystywany do szyfrowania danych (ang. bulk encryption) oraz algorytm do tworzenia MAC. Specyfikacja zestawu szyfrów (ang. cipher suite) przyjmuje jedną z przedstawionych form:

  TLS_<KEY>_WITH_<CIPHER>_<MAC>
  SSL_<KEY>_WITH_<CIPHER>_<MAC>

Znaczenie poszczególnych pól w nazwie zestawu szyfrów wyjaśnia poniższa tabela.

Nazwa	Opis
<KEY>	algorytm wymiany klucza wraz z metodą uwierzytelniania, np:. RSA, DH_RSA, DHE_RSA
<CIPHER>	algorytm symetryczny, długość klucza (jeżeli algorytm może korzystać z kluczy o różnej długości), tryb (np. CBC); przykłady: AES_256_CBC, RC4_128
<MAC>	jednokierunkowa funkcja skrótu wykorzystywana do wygenerowania MAC (ang. Message Authentication Code), np.: MD5, SHA, SHA256

Początkowo połączenie znajduje się w stanie TLS_NULL_WITH_NULL_NULL.

- Listę algorytmów kompresji danych. Najczęściej lista zawiera jeden element o wartości zero (null), co oznacza brak kompresji.
- W przypadku protokołu TLS (nie SSL), klient może przesłać dodatkowe dane w rozszerzeniach znajdujących się za listą algorytmów do kompresji danych.

2. Po wysłaniu wiadomości ClientHello, klient oczekuje na odpowiedź od serwera. Jeżeli serwer może zaakceptować jedną z propozycji klienta, to odpowiedź ma formę wiadomości ServerHello. Wiadomość zawiera:
   - Wersję protokołu SSL/TLS jaka będzie wykorzystywana do komunikacji.
   - 32-bajtową, losową liczbę. Wraz z losową liczbą klienta, posłuży ona do wygenerowania master secret.
   - Identyfikator sesji (opcjonalnie). Jeżeli wiadomość nie zawiera identyfikatora sesji, oznacza to, że serwer nie pozwala na jej wznowienie.
   - Zestaw algorytmów wybrany z listy przedstawionej przez klienta.
   - Metodę kompresji danych wybraną z listy przedstawionej przez klienta.

3. TLS (SSL) zapewnia trzy tryby uwierzytelniania: obu stron, serwera, brak uwierzytelniania. Tylko w dwóch pierwszych przypadkach połączenie jest zabezpieczone przed atakami typu man-in-the-middle. Serwer musi zostać uwierzytelniony, jeżeli wybrana metoda wymiany

klucza używa certyfikatów (wszystkie metody oprócz DH_anon i PSK). Uwierzytelnianie serwera odbywa się za pomocą wiadomości Certificate, którą serwer wysyła do klienta. Komunikat Certificate zawiera łańcuch certyfikatów X.509v3 – zaczynając od certyfikatu serwera, a kończąc na certyfikacie CA. Certyfikat CA może zostać pominięty, jeżeli znajduje się w posiadaniu klienta.

Metoda wymiany klucza ma doprowadzić do ustalenia wspólnego sekretu (premaster secret) i opcjonalnie umożliwić uwierzytelnienie jednej lub obu stron. Podstawowe metody wymiany klucza przedstawia poniższa tabela.

Metoda	**Opis**
RSA	Klient generuje pre-master secret, szyfruje pre-master secret za pomocą algorytmu RSA i klucza publicznego z certyfikatu serwera, po czym wysyła szyfrogram za pomocą wiadomości ClientKeyExchange. Metoda nie zapewnia PFC (ang. Perfect Forward Secrecy), tzn. w przypadku kompromitacji klucza serwera, wszystkie sesje utworzone za jego pomocą tracą poufność. Certyfikat musi umoŜliwiać uŜycie klucza publicznego do szyfrowania danych (jeŜeli rozszerzenie specyfikujące zastosowanie klucza występuje w certyfikacie).
DH_DSS DH_RSA	Metody określane jako fixed Diffie-Helman. Certyfikat serwera zawiera publiczny klucz DH ($g^x \ mod \ p$) wraz z parametrami wykorzystanymi do jego wygenerowania (g, p). Algorytm wykorzystany do podpisania certyfikatu jest określony w nazwie metody wymiany klucza (RSA lub DSA dla standardu

	DSS). Po odebraniu certyfikatu, klient weryfikuje go, wybiera prywatną wartość y i przesyła do serwera wiadomość ClientKeyExchange z publiczną wartością DH: $g^y \ mod \ p$. Po odebraniu wiadomości ClientKeyExchange, zarówno klient, jak i serwer mogą wygenerować pre-master secret (na podstawie $g^x \ mod \ p$ oraz $g^y \ mod \ p$). Metody zapewniają PFC. Certyfikat musi umoŜliwiać uŜycie klucza publicznego do celów uzgadniania klucza (jeŜeli rozszerzenie specyfikujące zastosowanie klucza występuje w certyfikacie).
DHE_DSS DHE_RSA	Metody określane jako ephemeral Diffie-Helman. Parametry algorytmu DH są wybierane losowo zarówno przez klienta, jak i serwera. Serwer przesyła swój klucz publiczny DH ($g^x \ mod \ p$) oraz parametry g, p w wiadomości ServerKeyExchange. Przesyłany klucz publiczny DH i parametry DH są podpisane za pomocą klucza prywatnego serwera. Algorytm wykorzystany do utworzenia podpisu jest określony w nazwie metody wymiany klucza (RSA lub DSA dla standardu DSS). Po odebraniu wiadomości Certificate i ServerKeyExchange, klient moŜe zweryfikować autentyczność klucza DH serwera za pomocą klucza publicznego (RSA lub DSA) z certyfikatu serwera. JeŜeli weryfikacja zakończy się pomyślnie, klient wybiera prywatną wartość y i przesyła do serwera wiadomość ClientKeyExchange z publiczną wartością DH: $g^y \ mod \ p$. Po odebraniu wiadomości ClientKeyExchange, zarówno klient, jak i serwer mogą wygenerować pre-master secret (na podstawie $g^x \ mod \ p$ oraz $g^y \ mod \ p$). Metody zapewniają PFC. Certyfikat musi umoŜliwiać uŜycie klucza publicznego do weryfikacji podpisów cyfrowych (jeŜeli rozszerzenie specyfikujące zastosowanie klucza występuje w certyfikacie).
DH_anon	Metoda nie zapewnia uwierzytelniania i nie wykorzystuje wiadomości Certificate. Publiczne

	parametry DH serwera są przesyłane w wiadomości ServerKeyExchange. Podobnie publiczne parametry DH klienta są przesyłane w wiadomości ClientKeyExchange. Po wymianie wiadomości, klient i serwer mogą utworzyć pre-master secret. Metoda zapewnia PFC, jest odporna na pasywny sniffing, ale nie na ataki typu man-in-the-middle.

RFC 4279 definiuje dodatkowo metody PSK (ang. Pre-shared Key), a RFC 4492 metody bazujące na krzywych eliptycznych (ang. Elliptic Curve Cryptography).

4. W przypadku metod wymiany klucza: DHE_RSA, DHE_DSS i DH_anon, serwer wysyła wiadomość ServerKeyExchange. Wiadomość zawiera publiczną wartość DH serwera ($g^x \ mod \ p$) oraz parametry: p (ang. prime modulus) oraz g (ang. generator). Metody: RSA, DH_RSA, DH_DSS nie wymagają wysłania wiadomości ServerKeyExchange, ponieważ wszystkie dane potrzebne do wymiany klucza (premaster secret) zostały wysłane w certyfikacie serwera.

5. Serwer może zażądać certyfikatu klienta. W tym przypadku po wiadomości ServerKeyExchange (lub Certificate w przypadku metod RSA, DH_RSA, DH_DSS) serwer wysyła CertificateRequest. Wiadomość CertificateRequest zawiera m.in. listę typów certyfikatów, jakie serwer może zaakceptować. Przykładowe typy to:
   - certyfikat zawierający klucz RSA,
   - certyfikat zawierający klucz DSA,
   - certyfikat zawierający statyczny klucz DH.

6. Wiadomość ServerHelloDone oznacza, że serwer zakończył etap związany z wymianą klucza (pre-master secret) i oczekuje na odpowiedź klienta. Wiadomość ServerHelloDone nie przenosi żadnych istotnych informacji.

7. Jeżeli serwer zażądał certyfikatu klienta, to klient powinien przesłać za pomocą wiadomości Certificate łańcuch certyfikatów X.509v3. Łańcuch zaczyna się od certyfikatu klienta, a kończy na certyfikacie CA. Jeżeli klient nie posiada certyfikatu, musi przesłać wiadomość z pustą listą certyfikatów. Od decyzji serwera zależy wówczas to, czy ustanawianie połączenia TLS będzie kontynuowane.

8. Klient wysyła wiadomość ClientKeyExchange. Wiadomość jest wysyłana zawsze:
    - po ServerHelloDone, jeżeli klient nie jest uwierzytelniany,
    - po Certificate, jeżeli serwer zażądał certyfikatu klienta.

Zawartość wiadomości zależy od wynegocjowanej metody uzgadniania klucza:

Metoda	**Zawartość wiadomości ClientKeyExchange**
RSA	Klient generuje pre-master secret (jest to pseudolosowe 48 bajtów danych). Pre-master secret jest szyfrowany za pomocą algorytmu RSA i klucza publicznego z certyfikatu serwera. Wiadomość ClientKeyExchange przenosi szyfrogram utworzony na podstawie premaster secret. Po odebraniu i odszyfrowaniu wiadomości ClientKeyExchange, zarówno klient jaki serwer są

	w posiadaniu pre-master secret.
DHE_DSS DHE_RSA	Klient wybiera prywatną wartość y i przesyła do serwera publiczną wartość DH: $g^y \ mod \ p$. Po odebraniu wiadomości ClientKeyExchange, klient i serwer są w posiadaniu publicznych wartości DH: $g^y \ mod \ p$ oraz $g^y \ mod \ p$. Na ich podstawie tworzą pre-master secret: $Z = (g^y \ mod \ p)^x \ mod \ p = (g^x \ mod \ p)^y \ mod \ p$
DH_DSS DH_RSA	Klient wybiera prywatną wartość y i przesyła do serwera publiczną wartość DH: $g^y \ mod \ p$. JeŜeli klient wysłał certyfikat do serwera i certyfikat zawiera publiczną wartość DH (statyczny klucz DH): $g^y \ mod \ p$, to przesyłana wiadomość jest pusta. Po odebraniu wiadomości ClientKeyExchange, klient i serwer są w posiadaniu publicznych wartości DH: $g^y \ mod \ p$ oraz $g^y \ mod \ p$. Na ich podstawie tworzą pre-master secret: $Z = (g^y \ mod \ p)^x \ mod \ p = (g^x \ mod \ p)^y \ mod \ p$

Na podstawie wynegocjowanego sekretu (pre-master secret) oraz losowych wartości przesłanych w wiadomościach ClientHello i ServerHello, klient i serwer tworzą master secret o rozmiarze 48 bajtów. Procedura generowania master secret różni się dla protokołów SSL i TLS, ale w obu przypadkach opiera się na funkcjach skrótu. Na podstawie master secret tworzony jest materiał kryptograficzny. Dopiero z materiału kryptograficznego wydzielane są bloki danych używane jako klucze. Obie strony stosują tą samą procedurę wygenerowania kluczy. W jej wyniku powstają następujące klucze:

Nazwa klucza	Opis
client_write_MA	Klucz używany do tworzenia MAC dla

C_key	wiadomości wysyłanych przez klienta.
server_write_MAC_key	Klucz używany do tworzenia MAC dla wiadomości wysyłanych przez serwer.
client_write_key	Klucz używany do szyfrowania wiadomości wysyłanych przez klienta. Serwer deszyfruje za jego pomocą odebrane wiadomości.
server_write_key	Klucz używany do szyfrowania wiadomości wysyłanych przez serwer. Klient deszyfruje za jego pomocą odebrane wiadomości.
client_write_IV	Wektor inicjalizacyjny klienta (opcjonalnie).
server_write_IV	Wektor inicjalizacyjny serwera (opcjonalnie).

9. Jeżeli klient wysłał wiadomość Certificate (klient jest uwierzytelniany), to kolejną wiadomością wysyłaną przez klienta jest CertificateVerify. Wiadomość CertificateVerify zawiera podpis cyfrowy utworzony na podstawie wszystkich wiadomości TLS wymienionych do tej pory między klientem i serwerem. Podpis jest tworzony za pomocą klucza prywatnego klienta. Ponieważ serwer jest w posiadaniu certyfikatu klienta (klucza publicznego), będzie w stanie zweryfikować podpis.

10. Klient wysyła wiadomość ChangeCipherSpec. Wiadomość przenosi jeden bajt danych (wartość 1), a jej zadaniem jest poinformowanie odbiorcy o zmianie zestawu algorytmów kryptograficznych i kluczy:

- Odbiorca wiadomości (serwer) musi natychmiast zmienić zestaw algorytmów dla wiadomości przychodzących.
- Nadawca wiadomości (klient) zmienia zestaw algorytmów dla wiadomości wysyłanych.

Zmiana jest dokonywana z obecnego zestawu algorytmów na zestaw wynegocjowany.

11. Klient wysyła wiadomość Finished.
Na podstawie master secret, etykiety „client finished" oraz wszystkich wiadomości TLS wymienionych do tej pory między klientem a serwerem tworzony jest skrót. Skrót jest transmitowany z zastosowaniem wynegocjowanych algorytmów kryptograficznych, tzn. z MAC i w formie zaszyfrowanej. Odbiorca wiadomości (serwer) musi zweryfikować jej poprawność.

12. Serwer wysyła wiadomość ChangeCipherSpec (jak w punkcie 10).

13. Serwer wysyła wiadomość Finished.
Na podstawie master secret, etykiety „server finished" oraz wszystkich wiadomości TLS wymienionych do tej pory między klientem a serwerem tworzony jest skrót. Skrót jest transmitowany z zastosowaniem wynegocjowanych algorytmów kryptograficznych, tzn. z MAC i w formie zaszyfrowanej. Odbiorca wiadomości (klient) musi zweryfikować jej poprawność.

14. Jeżeli wiadomości Finished zostaną uznane za poprawne, klient i serwer mogą rozpocząć wymianę informacji przez bezpieczne połączenie TLS.

1.3.4. Obsługa błędów w OpenSSL

OpenSSL jest biblioteką tak złożoną, że posiada osobną sekcję podręcznika systemowego (man 3ssl), a do obsługi błędów wykorzystuje własną bibliotekę (ERR). Gdy wywołanie funkcji OpenSSL zakończy się niepowodzeniem, błąd jest sygnalizowany za pomocą wartości zwracanej funkcji, a informacja o błędzie jest umieszczana w kolejce błędów. Każdy wątek posiada oddzielną kolejkę błędów typu FIFO (błędy są pobierane w kolejności ich wygenerowania). Podstawową informacją na temat błędu jest jego kod: 32-bitowa liczba, która ma znaczenie tylko dla OpenSSL. Kod błędu składa się z numeru biblioteki, kodu funkcji oraz kodu określającego przyczynę wystąpienia błędu. Dla wygody programistów, OpenSSL definiuje funkcję ERR_print_errors_fp(), która wysyła informacje na temat wszystkich błędów z kolejki wątku do strumienia fp. Wywołanie funkcji ERR_print_errors_fp() powoduje opróżnienie kolejki błędów danego wątku.

```
#include <openssl/err.h>

void ERR_print_errors_fp(FILE *fp);
```

Przed użyciem funkcji ERR_print_errors_fp(), warto załadować tekstowe opisy błędów do pamięci. W tym celu należy wywołać funkcję ERR_load_crypto_strings():

```
#include <openssl/ssl.h>

void SSL_load_error_strings(void); /* Dla libcrypto i
libssl. */
```

Tekstowe opisy błędów można zwolnić z pamięci za pomocą funkcji ERR_free_strings():

```
#include <openssl/err.h>

void ERR_free_strings(void);
```

1.3.5. Biblioteka libssl

Biblioteka libssl implementuje protokoły SSLv2, SSLv3 oraz TLSv1. API biblioteki jest bardzo rozległe - obejmuje ponad 200 funkcji. W celu zainicjalizowania biblioteki należy wywołać funkcję SSL_library_init(), która rejestruje nazwy algorytmów kryptograficznych i funkcji skrótu:

```
#include <openssl/ssl.h>

int SSL_library_init(void);
```

Kluczową kwestią jest zainicjalizowanie generatora liczb pseudolosowych. Jedną z metod stosowanych w systemie Linux jest wykorzystanie pliku /dev/random jako źródła entropii:

```
/*
 * Inicjalizacja generatora liczb pseudolosowych za
pomocą
 * 32 bajtów z pliku /dev/random:
 */
RAND_load_file(“/dev/random”, 32);
```

Plik /dev/random gromadzi entropię na podstawie „szumu środowiska” (informacji ze sterowników urządzeń itp.). Plik ten jest źródłem danych losowych o bardzo wysokiej jakości. Jeśli pula losowa jest pusta, odczyt z /dev/random będzie wstrzymany do czasu zebrania dodatkowego „szumu”. Alternatywnym rozwiązaniem jest wykorzystanie pliku /dev/urandom. Odczyt z niego jest operacją nieblokującą - zawsze zwracana jest żądana

liczba bajtów, ale plik ten jest źródłem entropii gorszej jakości.

Biblioteka libssl definiuje wiele struktur związanych z implementacją protokołów SSL/TLS. Do najważniejszych z nich należą:

- SSL_METHOD – opisuje konkretną implementację protokołu oraz rolę jaką będzie pełnił proces korzystający z tej struktury (rolę klienta lub serwera). Struktura SSL_METHOD przechowuje wskaźniki na funkcje dla określonej wersji protokołu i jest wymagana do utworzenia kontekstu (SSL_CTX). API OpenSSL udostępnia szereg funkcji, za pomocą których można uzyskać odpowiednio zainicjalizowany obiekt SSL_METHOD. Funkcje te mają postać:

```
<wersja>_<rola>_method()
```

np.:

```
#include <openssl/ssl.h>

const SSL_METHOD *TLSv1_client_method(void);
const SSL_METHOD *SSLv3_server_method(void);
```

- SSL_CTX – kontekst zawierający domyślną konfigurację dla tworzonych połączeń SSL/TLS. Kontekst pozwala zdefiniować wersję protokołu, listę zestawów algorytmów kryptograficznych, repozytorium certyfikatów, klucz prywatny, itp. Na podstawie kontekstu tworzone są obiekty SSL reprezentujące połączenia. Funkcje modyfikujące kontekst posiadają prefiks SSL_CTX_. W celu utworzenia kontekstu należy wywołać funkcję SSL_CTX_new(). Funkcja przyjmuje jeden argument – wartość zwracaną przez funkcje <wersja>_<rola>_method():

```
#include <openssl/ssl.h>

SSL_CTX *SSL_CTX_new(SSL_METHOD *meth);
```

Zwalnianie kontekstu odbywa się za pomocą funkcji SSL_CTX_free():

```
#include <openssl/ssl.h>

void SSL_CTX_free(SSL_CTX *ctx);
```

- SSL – struktura reprezentująca konkretne połączenie SSL/TLS. Zmienną typu SSL można utworzyć na podstawie kontekstu, za pomocą funkcji SSL_new():

```
#include <openssl/ssl.h>

SSL *SSL_new(SSL_CTX *ctx);
```

Utworzony obiekt SSL dziedziczy wszystkie ustawienia kontekstu. Co więcej obiekt SSL zawiera wskaźnik na strukturę SSL_CTX, na podstawie której został utworzony. Zmiany w kontekście mogą spowodować zmiany we wszystkich połączeniach (obiektach) SSL ustanowionych za pomocą danego kontekstu. W celu wprowadzenia zmian dla konkretnego połączenia, można posłużyć się funkcjami z prefiksem SSL_.

Konfigurację kontekstu najwygodniej jest przeprowadzić przed ustanowieniem połączenia SSL/TLS. Wówczas wszystkie tworzone połączenia dziedziczą ustawienia kontekstu.

1.3.6. Konfiguracja kontekstu

Jeżeli wybrana metoda wymiany klucza wymaga uwierzytelniania, to po stronie serwera niezbędne jest załadowanie certyfikatu, który będzie wysyłany klientom podczas ustanawiania połączenia. Zazwyczaj certyfikat

ładuje się z pliku w formacie PEM, zawierającego łańcuch certyfikatów (od certyfikatu serwera, aż do certyfikatu głównego CA). Do załadowania łańcucha certyfikatów używa się funkcji SSL_CTX_use_certificate_chain_file().

```
#include <openssl/ssl.h>

int SSL_CTX_use_certificate_chain_file(SSL_CTX* ctx,
const char* file);
```

Parametr	Opis
ctx	kontekst połączenia
file	plik z certyfikatami

Ponadto, konieczne jest załadowanie klucza prywatnego odpowiadającego kluczowi publicznemu z certyfikatu serwera. Załadowanie klucza prywatnego można przeprowadzić za pomocą funkcji SSL_CTX_use_PrivateKey_file().

```
#include <openssl/ssl.h>

int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char
*file,
    int type);
```

Parametr	Opis
ctx	kontekst połączenia
file	plik z kluczem prywatnym
type	format pliku: SSL_FILETYPE_PEM lub

	SSL_FILETYPE_ASN1

W celu weryfikacji certyfikatu serwera konieczne jest załadowanie certyfikatów zaufanych. Ładowanie bazy certyfikatów dokonuje się przy pomocy funkcji SSL_CTX_load_verify_locations().

```
#include <openssl/ssl.h>

int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const
char *CAfile,
    const char *CApath);
```

Parametr	Opis
ctx	kontekst połączenia
CAfile	plik zawierający zaufane certyfikaty CA w formacie PEM
CApath	katalog zawierający zaufane certyfikaty CA

Podczas wywołania funkcji SSL_CTX_load_verify_locations() co najmniej jeden z argumentów CAfile, CApath powinien być różny od NULL.

Katalog zdefiniowany przez parametr CApath wymaga zainicjalizowania za pomocą programu c_rehash. Program c_rehash przyjmuje jako argument ścieżkę do katalogu zawierającego certyfikaty CA i tworzy w katalogu dowiązania symboliczne do certyfikatów. Dowiązania są wykorzystywane przez OpenSSL w celu wyszukiwania i ładowania certyfikatów.

Plik określony przez parametr CAfile jest ładowany do pamięci w momencie wywołania funkcji SSL_CTX_load_verify_locations(), natomiast certyfikaty z katalogu CApath są ładowane przez OpenSSL w razie potrzeby.

1.3.7. Bezpieczne połączenie po stronie klienta

W celu ustanowienia połączenia SSL/TLS z serwerem za pomocą funkcji systemowych omówionych w podrozdziale 1.3.1, konieczne jest wykonanie następujących kroków:

- inicjalizacja biblioteki OpenSSL,
- utworzenie i konfiguracja kontekstu połączenia,
- utworzenie gniazda przy pomocy funkcji socket(),
- nawiązanie połączenia TCP za pomocą funkcji connect(),
- utworzenie struktury SSL na podstawie kontekstu,
- powiązanie deskryptora gniazda ze strukturą SSL,
- rozpoczęcie negocjacji SSL/TLS za pomocą funkcji SSL_connect().

Poniżej przedstawiony jest fragment kodu źródłowego programu klienta, odpowiedzialny za nawiązanie połączenia TLS z serwerem.

```
int sockfd; /* Desktryptor gniazda. */
int retval; /* Wartość zwracana przez funkcje. */
struct sockaddr_in remote_addr;/* Gniazdowa struktura
adresowa. */
socklen_t addr_len; /* Rozmiar struktury w bajtach. */
SSL_CTX *ctx; /* Kontekst SSL. */
SSL *ssl;

/* Wczytanie tekstowych opisów błędów: */
SSL_load_error_strings();

/* Inicjalizacja biblioteki - wczytanie nazw algorytmów
```

```
szyfrujących
 * i funkcji skrótu: */
SSL_library_init();

/*
 * Inicjalizacja generatora liczb pseudolosowych za
pomocą pliku
 * /dev/urandom:
 */
RAND_load_file("/dev/urandom", 128);

/* Utworzenie kontekstu połączenia dla protokołu TLSv1:
*/
ctx = SSL_CTX_new(TLSv1_client_method());
if (ctx == NULL) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/*
 * Konfiguracja kontekstu…
 */

/*
 * Utworzenie struktury SSL. Konfiguracja jest kopiowana
z kontekstu.
 */
ssl = SSL_new(ctx);
if (ssl == NULL) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/* Utworzenie gniazda dla protokolu TCP: */
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
if (sockfd == -1) {
    perror("socket()");
    exit(EXIT_FAILURE);
}

/* Powiązanie struktury SSL z deskryptorem pliku: */
retval = SSL_set_fd(ssl, sockfd);
if (retval == 0) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/* Inicjalizacja struktury adresowej serwera
(remote_addr) ... */

/* Nawiązanie połączenia TCP: */
if(connect(sockfd,(const struct
sockaddr*)&remote_addr,addr_len)==-1) {
    perror("connect()");
    exit(EXIT_FAILURE);
}

/* Inicjacja połączenia TLS (TLS handshake): */
retval = SSL_connect(ssl);
if (retval ⩽ 0) {
    retval = SSL_get_error(ssl, retval);
    switch (retval) {
        case SSL_ERROR_ZERO_RETURN:
            fprintf(stderr, "SSL_connect(): closure
alert!\n");
            exit(EXIT_FAILURE);
        case SSL_ERROR_SSL:
            fprintf(stderr, "SSL_connect(): SSL
error!\n");
            exit(EXIT_FAILURE);
```

```
        default:
            fprintf(stderr, "SSL_connect(): error
(other)!\n");
            exit(EXIT_FAILURE);
    }
}
```

Rozpoczęcie procesu ustanawiania połączenia SSL/TLS odbywa się za pomocą funkcji SSL_connect():

```
#include <openssl/ssl.h>

int SSL_connect(SSL *ssl);
```

Funkcja SSL_connect() zwraca 1 w przypadku ustanowienia połączenia SSL/TLS, 0 w przypadku błędu ze względu na specyfikację protokołu, -1 w przypadku błędu krytycznego związanego z implementacją lub połączeniem.

1.3.8. Bezpieczne połączenie po stronie serwera

Aby za pomocą funkcji systemowych ustanowić bezpieczne połączenie po stronie serwera, konieczne jest wykonanie następujących kroków:

- inicjalizacja biblioteki OpenSSL,
- utworzenie i konfiguracja kontekstu połączenia,
- utworzenie gniazda przy pomocy funkcji socket(),
- powiązanie gniazda z lokalnym adresem IP i numerem portu,
- przekształcenie gniazda serwera w gniazdo nasłuchujące,
- akceptacja połączenia TCP za pomocą funkcji accept(),
- utworzenie struktury SSL na podstawie kontekstu,
- powiązanie deskryptora gniazda połączonego ze strukturą SSL,
- akceptacja połączenia SSL/TLS za pomocą funkcji SSL_accept().

Poniżej przedstawiony jest fragment kodu źródłowego programu serwera, odpowiedzialny za nawiązanie połączenia TLS z klientem.

```
/* Deskryptory dla gniazda nasłuchującego i połączonego:
*/
int listenfd, connfd;

int retval; /* Wartość zwracana przez funkcje. */

/* Gniazdowe struktury adresowe (dla klienta i serwera):
*/
struct sockaddr_in client_addr, server_addr;

/* Rozmiar struktur w bajtach: */
socklen_t client_addr_len, server_addr_len;
SSL_CTX *ctx; /* Kontekst SSL. */
SSL *ssl;

/* Wczytanie tekstowych opisów błędów: */
SSL_load_error_strings();

/* Inicjalizacja biblioteki - wczytanie nazw algorytmów
szyfrujących
 * i funkcji skrótu: */
SSL_library_init();

/*
 * Inicjalizacja generatora liczb pseudolosowych za
pomocą pliku
 * /dev/urandom:
 */
RAND_load_file("/dev/urandom", 128);

/* Utworzenie kontekstu połączenia dla protokołu TLSv1:
```

```
*/
ctx = SSL_CTX_new(TLSv1_server_method());
if (ctx == NULL) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/*
 * Konfiguracja kontekstu…
 */

/* Utworzenie gniazda dla protokołu TCP: */
listenfd = socket(PF_INET, SOCK_STREAM, 0);
if (listenfd == -1) {
    perror("socket()");
    exit(EXIT_FAILURE);
}

/* Inicjalizacja struktury adresowej serwera
(server_addr) ... */

/* Powiązanie adresu IP i numeru portu z gniazdem: */
if (bind(listenfd,(struct
sockaddr*)&server_addr,server_addr_len)==-1)
{
    perror("bind()");
    exit(EXIT_FAILURE);
}

/* Przekształcenie gniazda serwera w gniazdo
nasłuchujące: */
if (listen(listenfd, 5) == -1) {
    perror("listen()");
    exit(EXIT_FAILURE);
}
```

```
/* Akceptacja połączenia TCP: */
client_addr_len = sizeof(client_addr);
connfd = accept(listenfd,
    (struct sockaddr*)&client_addr, &client_addr_len);
if (connfd == -1) {
    perror("accept()");
    exit(EXIT_FAILURE);
}

/*
 * Utworzenie struktury SSL. Konfiguracja jest kopiowana
z kontekstu.
 */
ssl = SSL_new(ctx);
if (ssl == NULL) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/* Powiązanie struktury SSL z deskryptorem gniazda
połączonego: */
retval = SSL_set_fd(ssl, connfd);
if (retval == 0) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

/* Akceptacja połączenia TLS: */
retval = SSL_accept(ssl);
if (retval <= 0) {
    retval = SSL_get_error(ssl, retval);
    switch (retval) {
        case SSL_ERROR_ZERO_RETURN:
            fprintf(stderr, "SSL_accept(): closure
```

```
alert!\n");
            exit(EXIT_FAILURE);
        case SSL_ERROR_SSL:
            fprintf(stderr, "SSL_accept(): SSL
error!\n");
            exit(EXIT_FAILURE);
        default:
            fprintf(stderr, "SSL_accept(): error
(other)!\n");
            exit(EXIT_FAILURE);
    }
}
```

Po wywołaniu funkcji SSL_accept(), serwer oczekuje na wiadomość ClientHello. Otrzymanie wiadomości ClientHello jest sygnałem rozpoczęcia przez klienta procesu ustanawiania połączenia TLS/SSL.

```
#include <openssl/ssl.h>

int SSL_accept(SSL *ssl);
```

Funkcja SSL_accept() zwraca 1 w przypadku ustanowienia połączenia SSL/TLS, 0 w przypadku błędu ze względu na specyfikację protokołu, -1 w przypadku błędu krytycznego związanego z implementacją lub połączeniem.

1.3.9. Operacje I/O

Operacje wejścia/wyjścia są przeprowadzane za pomocą funkcji SSL_read() oraz SSL_write().

Funkcja SSL_read() odczytuje dane z bezpiecznego połączenia SSL/TLS:

```
#include <openssl/ssl.h>

int SSL_read(SSL *ssl, void *buf, int num);
```

Parametr	Opis
ssl	struktura SSL reprezentująca nawiązane połączenie
buf	bufor, do którego zostaną zapisane dane
num	ilość bajtów do przeczytania

Funkcja SSL_read() zwraca liczbę odczytanych bajtów, 0 w przypadku zamknięcia połączenia, -1 w przypadku błędu.

Funkcja SSL_write() zapisuje dane do bezpiecznego połączenia SSL/TLS:

```
#include <openssl/ssl.h>

int SSL_read(SSL *ssl, void *buf, int num);
```

Parametr	Opis
ssl	struktura SSL reprezentująca nawiązane połączenie
buf	bufor z danymi do zapisu
num	ilość bajtów do zapisania

Funkcja SSL_write() zwraca liczbę zapisanych bajtów, 0 w przypadku zamknięcia połączenia, -1 w przypadku błędu.

1.4. Cel laboratorium

Celem laboratorium jest zapoznanie się z protokołami SSL/TLS oraz interfejsem programistycznym OpenSSL. Po zrealizowaniu laboratorium powinieneś:

- rozumieć proces ustanawiania połączenia SSL/TLS,
- znać podstawowe funkcje API OpenSSL dla protokołów SSL/TLS,
- umieć korzystać z programu openssl w celu generowania kluczy RSA, tworzenia żądań certyfikacyjnych, wystawiania certyfikatów,
- umieć ustanawiać bezpieczne połączenia SSL/TLS - z uwierzytelnianiem serwera lub obu stron biorących udział w komunikacji.

2. Przebieg laboratorium

Druga część instrukcji zawiera zadania do praktycznej realizacji, które demonstrują zastosowanie technik z omawianego zagadnienia.

2.1. Zadanie 1. Gniazda TCP

Zadanie polega na analizie kodu przykładowych programów klienta i serwera TCP. Klient nawiązuje połączenie z serwerem – adres i numer portu serwera podane są w argumentach wywołania programu. Po zaakceptowaniu połączenia, serwer wysyła do klienta wiadomość „Laboratorium PUS.”, zamyka gniazdo połączone i nasłuchuje na kolejne połączenia.

W celu uruchomienia programów należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować programy źródłowe do postaci binarnej:

```
$ gcc tcp_client.c -o tcp_client
$ gcc tcp_server.c -o tcp_server
```

Albu użyć makefile w katalogu wyżej, wtedy pliki zostaną skompilowane w podkatalogu bin:

```
$ make
```

3. Uruchomić program serwera podając numer portu TCP, na którym ma nasłuchiwać:

```
$ ./tcp_server <numer portu>
```

4. Uruchomić sniffer (np. tshark) z opcjami umożliwiającymi przechwytywanie pakietów TCP:

```
$ sudo tshark -i <interfejs>
```

5. Uruchomić program klienta podając adres IP i numer portu serwera:

```
$ ./tcp_client <adres IP> <numer portu>
```

6. Zaobserwować przebieg połączenia za pomocą sniffera.

2.2. Zadanie 2. Połączenie TLS

Celem zadania jest analiza kodu źródłowego programów wymieniających dane za pomocą protokołu TLS. Klient nawiązuje połączenie z serwerem, po czym oczekuje na wiadomość „Laboratorium PUS.”. Serwer używa klucza RSA, który należy wcześniej wygenerować poleceniem:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout server.pem -out server.pem
```

W celu uruchomienia programów należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować programy źródłowe do postaci binarnej:

   ```
   $ gcc tls_client.c -lssl -lcrypto -o tls_client
   $ gcc tls_server.c -lssl -lcrypto -o tls_server
   ```

3. Upewnić się, że w katalogu z plikiem wykonywalnym tls_server znajduje się pliki server.pem.
4. Uruchomić program serwera podając numer portu TCP, na którym ma nasłuchiwać:

   ```
   $ ./tls_server <numer portu>
   ```

5. Uruchomić sniffer (np. tshark):

   ```
   $ sudo tshark -i <interfejs> -d tcp.port==<port serwera>,ssl port <port serwera>
   ```

   Opcja -V wyświetla szczegóły na temat nagłówków wszystkich protokołów. W przypadku użycia opcji -V najlepiej jest przekierować standardowe wyjście programu do pliku i później przeanalizować jego zawartość (proszę nie używać opcji -w <nazwa pliku>):

   ```
   $ sudo tshark -i <interfejs> -d tcp.port==<port serwera>,ssl  port <port serwera> -V > ”<nazwa pliku>”
   ```

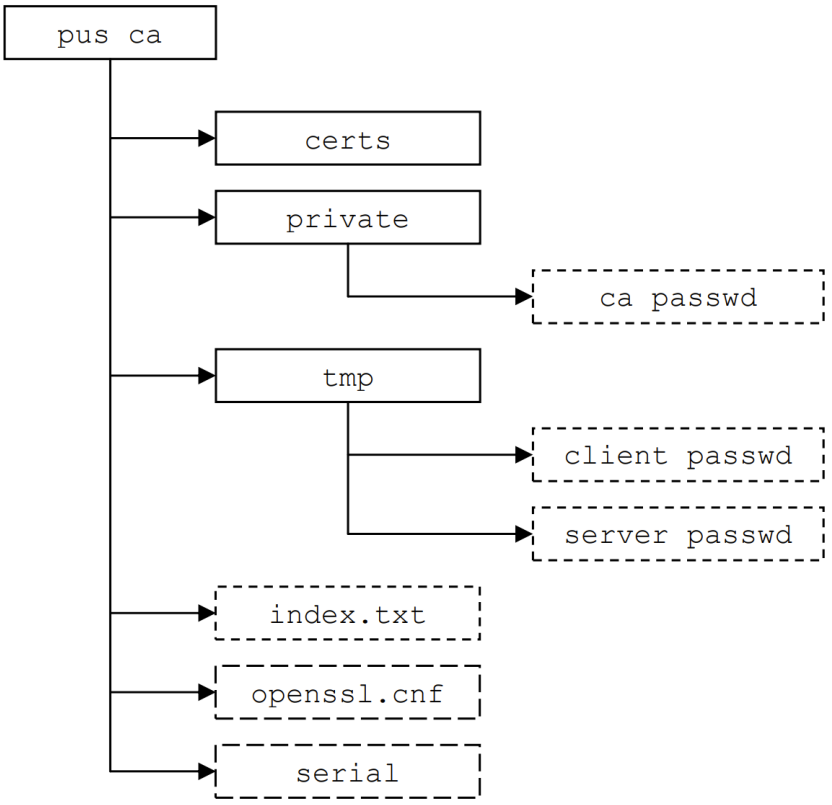
6. Uruchomić program klienta podając adres IP i numer portu serwera:

   ```
   $ ./tls_client <adres IP> <numer portu>
   ```

7. Zaobserwować przebieg połączenia za pomocą sniffera. Jakie wiadomości TLS są wymieniane podczas ustanawiania połączenia? Jaka wiadomość kończy sesję TLS?

2.3. Zadanie 3. Centrum certyfikacji

Kolejne zadania wymagają certyfikatów klucza publicznego dla klienta i serwera. W celu ich wygenerowania proszę posłużyć się centrum certyfikacji (ang. certificate authority) przygotowanym w archiwum pus_ca.tar.



Rys. 2. Organizacja archiwum pus_ca.tar.

Archiwum pus_ca.tar posiada następującą organizację (rys. 2):

- Katalog certs będzie przechowywał kopie wygenerowanych certyfikatów klucza publicznego. Nazwy certyfikatów mają następującą formę: <numer seryjny certyfikatu>.pem.

- Katalog private zawiera plik ca_password z hasłem zabezpieczającym klucz prywatny CA.
- Katalog tmp zawiera pliki z hasłami zabezpieczającymi klucz prywatny klienta i serwera.
- Plik index.txt stanowi bazę danych wystawionych certyfikatów i jest wykorzystywany podczas ich unieważniania.
- Plik openssl.cnf zawiera konfigurację centrum certyfikacji, a plik serial - numer seryjny kolejnego certyfikatu.

W celu wygenerowania certyfikatów należy:

1. Rozpakować archiwum pus_ca.tar do bieżącego katalogu roboczego.
2. Przejść do katalogu pus_ca:

   ```
   $ cd pus_ca
   ```

3. Uruchomić skrypt create_certs.sh:

   ```
   $ ./create_certs.sh
   ```

W wyniku działania skryptu powstają następujące pliki:

- ca_cert.pem – certyfikat centrum certyfikacji (typu self-signed),
- private/ca_keypair.pem – para kluczy RSA centrum certyfikacji; plik jest zaszyfrowany algorytmem AES, ponieważ zawiera klucz prywatny; hasło wymagane do odszyfrowania („secret”) znajduje się w pliku private/ca_passwd,
- tmp/server_keypair.pem – para kluczy RSA serwera; plik jest zaszyfrowany algorytmem AES, ponieważ zawiera klucz prywatny; hasło wymagane do odszyfrowania („server”) znajduje się w pliku tmp/server_passwd,

- tmp/server_req.pem – żądanie certyfikacyjne serwera,
- tmp/server_cert.pem – certyfikat serwera wystawiony przez centrum certyfikacji (podpisany kluczem prywatnym CA z pliku private/ca_keypair.pem),
- tmp/server_chain.pem – łańcuch certyfikatów serwera; zawiera certyfikat serwera oraz certyfikat CA,
- tmp/client_keypair.pem – para kluczy RSA klienta; plik jest zaszyfrowany algorytmem AES, ponieważ zawiera klucz prywatny; hasło wymagane do odszyfrowania („client”) znajduje się w pliku tmp/client_passwd,
- tmp/server_req.pem – żądanie certyfikacyjne klienta,
- tmp/server_cert.pem – certyfikat klienta wystawiony przez centrum certyfikacji (podpisany kluczem prywatnym CA z pliku private/ca_keypair.pem),
- tmp/client_chain.pem – łańcuch certyfikatów klienta; zawiera certyfikat klienta oraz certyfikat CA,

Proszę przeanalizować plik create_certs.sh odpowiedzialny za wygenerowanie kluczy i certyfikatów.

2.4. Zadanie 4. Połączenie TLS z uwierzytelnianiem serwera

Zadanie polega na analizie kodu źródłowego klienta i serwera TLS. Programy wykorzystują metodę wymiany klucza opierającą się na certyfikatach X509. Metoda umożliwia uwierzytelnianie obu stron biorących udział w komunikacji, ale w zadaniu została wykorzystana tylko do uwierzytelniania serwera.

W celu uruchomienia programów należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).

2. Skompilować programy źródłowe do postaci binarnej:

   ```
   $ gcc tls_client_check.c -lssl -lcrypto -o
   tls_client_check
   $ gcc tls_server_check.c -lssl -lcrypto -o
   tls_server_check
   ```

3. Skopiować klucz prywatny serwera (pus_ca/tmp/server_keypair.pem) oraz plik z łańcuchem certyfikatów serwera (pus_ca/tmp/server_chain.pem) do katalogu z plikiem wykonywalnym serwera.

4. W katalogu z plikiem wykonywalnym klienta utworzyć katalog cert_dir:

   ```
   $ mkdir cert_dir
   ```

5. Uruchomić program serwera podając numer portu TCP, na którym ma nasłuchiwać:

   ```
   $ ./tls_server_check <numer portu>
   ```

   Uwaga! Trzeba będzie wpisać hasło z pliku pus_ca/tmp/server_passwd

6. Uruchomić sniffer (tshark) z opcjami umożliwiającymi analizę nagłówków TLS:
7. Uruchomić program klienta podając adres IP i numer portu serwera:

   ```
   $ ./tls_client_check <adres IP> <numer portu>
   ```

8. Zaobserwować przebieg połączenia za pomocą sniffera. Jakie wiadomości TLS są wymieniane podczas

ustanawiania połączenia? Połączenie nie powinno się powieść, bo do cert_dir nie skopiowaliśmy certyfikatu, którym jest podpisany certyfikat serwera.
9. Do katalogu cert_dir proszę skopiować certyfikat CA (pus_ca/ca_cert.pem).
10. Utworzyć link symboliczny, który pozwoli na wyszukiwanie i pobranie certyfikatu przez programy korzystające z OpenSSL:

    ```
    $ c_rehash cert_dir
    ```

11. Ponownie uruchomić klienta, tym razem walidacja certyfikatu powinna przejść pomyślnie.

2.5. Zadanie 5. Połączenie TLS z uwierzytelnianiem obu stron

Proszę zmodyfikować kod źródłowy programów z zadania 4 w taki sposób, aby podczas ustanawiania połączenia TLS, uwierzytelniany był nie tylko serwer, ale również klient.

W celu uruchomienia programów należy wykonać następujące czynności:

1. Skopiować pliki tls_client_check.c i tls_server_check.c do odpowiednio tls_client_both.c i tls_server_both.c (bo wtedy makefile je łyknie)

2. Wzorując się na linijkach z pliku tls_server_both.c odpowiedzialnych za ładowanie server_chain.pem i server_keypair.pem, zaimplementować to w tls_client_both.c tak aby ładowały pliki odpowiednio client_chain.pem i client_keypair.pem.

3. Wzorując się na linijkach z pliku tls_client_both.c odpowiedzialnych za ustawienie katalogu z zaufanymi

certyfikatami oraz ustawienie procedury callback weryfikacji certyfikatu zaimplementować to w tls_server_both.c.

4. Skompilować programy źródłowe do postaci binarnej:

   ```
   $ gcc tls_client_both.c -lssl -lcrypto -o tls_client_both
   $ gcc tls_server_both.c -lssl -lcrypto -o tls_server_both
   ```

5. Wykonać krotki 3-11 z zadania 4 dla klienta i serwera pamiętając, że certyfikaty dla klienta są w plikach client_chain.pem i client_keypair.pem. Klient i serwer mogą współdzielić ten sam katalog cert_dir z zadania 4.

6. Zaobserwować przebieg połączenia za pomocą sniffera. Jakie różnice występują w stosunku do wymiany komunikatów między programami z zadania 5? **Uwaga!** Teraz klient też zapyta o hasło z pliku pus_ca/tmp/client_passwd.

3. Opracowanie i sprawozdanie

Realizacja laboratorium pt. „Bezpieczna komunikacja w OpenSSL” polega na wykonaniu wszystkich zadań programistycznych podanych w drugiej części tej instrukcji. Wynikiem wykonania powinno być sprawozdanie w formie wydruku papierowego dostarczonego na kolejne zajęcia licząc od daty laboratorium, kiedy zadania zostały zadane.

Sprawozdanie powinno zawierać:

- opis metodyki realizacji zadań (system operacyjny, język programowania, biblioteki, itp.),

- algorytmy wykorzystane w zadaniach (zwłaszcza, jeśli zastosowane zostały rozwiązania nietypowe),
- opisy napisanych programów wraz z opcjami,
- trudniejsze kawałki kodu, opisane tak, jak w niniejszej instrukcji,
- uwagi oceniające ćwiczenie: trudne/łatwe, nie/realizowalne, nie/wymagające wcześniejszej znajomości zagadnień (wymienić jakich),
- wskazówki dotyczące ewentualnej poprawy instrukcji celem lepszego zrozumienia sensu oraz treści zadań.