Instytut Teleinformatyki

Wydział Inżynierii Elektrycznej i Komputerowej
Politechnika Krakowska

programowanie usług sieciowych

„Kryptografia w OpenSSL”

laboratorium: 09
system operacyjny: Linux

Kraków, 2009

Spis treści

1. Wiadomości wstępne

Pierwsza część niniejszej instrukcji zawiera podstawowe wiadomości teoretyczne dotyczące **biblioteki *OpenSSL*** i programowania aplikacji wykorzystujących zaimplementowane przez nią algorytmy kryptograficzne. Poznanie tych wiadomości umożliwi prawidłowe zrealizowanie praktycznej części laboratorium.

1.1. Tematyka laboratorium

Tematyką laboratorium jest programowanie aplikacji wykorzystujących algorytmy kryptograficzne zaimplementowane przez bibliotekę *OpenSSL*. Pierwsza wersja biblioteki *OpenSSL* (0.9.1c) powstała w 1998 roku, na podstawie projektu *SSLeay*. *OpenSSL* składa się z trzech zasadniczych części:

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

Biblioteka SSL implementuje protokoły *Secure Sockets Layer* (wersje 2 i 3) oraz TLS v1. Biblioteka kryptograficzna jest wykorzystywana przez bibliotekę SSL, a jej funkcjonalność obejmuje m.in.:

- szyfrowanie symetryczne,
- kryptografię klucza publicznego,
- uzgadnianie klucza,
- obsługę certyfikatów,
- skróty wiadomości i kody uwierzytelniające,
- generowanie liczb pseudo-losowych.

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.

OpenSSL jest obecnie najpopularniejszą, darmową biblioteką kryptograficzną dla języków C/C++. Biblioteka ta jest dostępna zarówno dla systemów Linux, jak i Windows. Wiele aplikacji wykorzystuje możliwości *OpenSSL*. Przykładem są: *OpenSSH*, *Stunnel*, czy moduł *Apacha – mod_ssl*. Wymagają one obecności biblioteki podczas kompilacji. Inne aplikacje, np.: *MySQL*, mogą zostać opcjonalnie skompilowane i skonfigurowane ze wsparciem dla *OpenSSL*.

1.2. Zagadnienia do przygotowania

Przed przystąpieniem do realizacji laboratorium należy zapoznać się z zagadnieniami dotyczącymi kryptografii i algorytmów kryptograficznych: [1, 2]

- szyfrowanie symetryczne i asymetryczne
- funkcje skrótu
- kody uwierzytelniające HMAC [3]
- podpisy cyfrowe
- szyfrowanie strumieniowe i blokowe
- zasada działania algorytmu RSA
- kodowanie DER i PEM

Ponadto, wymagana jest znajomość biblioteki kryptograficznej *libcrypto* w zakresie:

- obsługi błędów [4]
- używania obiektów BIO (ang. *Basic Input/Output*) [5]
- interfejsu EVP [6]
- interfejsu algorytmu RSA [7]
- interfejsu HMAC [8]
- obsługi formatu PEM [9]
- generowania liczb pseudolosowych [10]

Literatura:

[1] D.R. Stinson, *„Kryptografia w teorii i w praktyce”*, WNT.

[2] Bruce Schneier, *„Applied cryptography, Second Edition: Protocols, Algorithms and Source Code in C”*, *John Wiley & Sons*

[3] IETF, RFC 2104, *„Keyed-Hashing for Message Authentication”*

[4] MAN (3ssl), *„err”, „ERR_get_error”, „ERR_error_string”, „ERR_load_crypto_strings”*

[5] MAN (3ssl), *„bio”, „BIO_f_cipher”, „BIO_new”, „BIO_s_file”, „BIO_push”, „BIO_read”, „BIO_ctrl”, „BIO_should_retry”*

[6] MAN (3ssl), *„evp”, „EVP_EncryptInit”, „EVP_DigestInit”*

[7] MAN (3ssl), *„rsa”, „RSA_generate_key”, „RSA_check_key”, „RSA_sign”, „RSA_new”*

[8] MAN (3ssl), *„hmac”*

[9] MAN (3ssl), *„pem”* (*PEM_write_RSAPrivateKey*, *PEM_read_RSAPrivateKey*, *PEM_write_RSAPublicKey*, *PEM_read_RSAPublicKey*)

[10] MAN (3ssl), *„rand”, „RAND_load_file”*

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

[12] David Hook, *„Kryptografia w Javie. Od podstaw”, Helion*

1.3. Opis laboratorium

1.3.1. Obsługa błędów

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 z kolejki 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. W celu wydobycia tych informacji z kodu błędu można posłużyć się makrami `ERR_GET_LIB()`, `ERR_GET_FUNC()`, `ERR_GET_REASON()`.

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

int ERR_GET_LIB(unsigned long e);
int ERR_GET_FUNC(unsigned long e);
int ERR_GET_REASON(unsigned long e);
```

Podstawowe funkcje obsługi błędów przedstawione są na następującym listingu.

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

unsigned long ERR_get_error(void);
unsigned long ERR_peak_error(void);

unsigned long ERR_get_error_line(const char **file, int* line);
unsigned long ERR_peak_error(const char **file, int* line);

unsigned long ERR_get_error_line_data(const char **file, int* line,
                                      const char **data, int* flags);
unsigned long ERR_peak_error_line_data(const char **file, int* line,
                                       const char **data, int* flags);
```

Funkcje `get` pobierają informacje na temat błędu (informacje te są usuwane z kolejki). Funkcje `peak` pobierają informacje na temat błędu, ale nie usuwają ich z kolejki. Wszystkie wymienione funkcje zwracają kod błędu lub wartość zero, jeżeli kolejka błędów jest pusta. Parametry funkcji omówione są w poniższej tabeli.

Parametr	Opis
`file`	wskaźnik na string, który będzie przechowywał nazwę pliku źródłowego (z reguły na podstawie makra `__FILE__`); zwró-

	cony string nie powinien być modyfikowany
line	wskaźnik na zmienną, która będzie przechowywać numer linii w pliku (na podstawie makra __LINE__)
data	otrzyma wskaźnik na dodatkowe informacje o błędzie; wskaźnik nie może być modyfikowany
flags	maska bitowa, która określa jak należy traktować parametr data; jeżeli ustawiona jest flaga ERR_TXT_STRING, to *data jest stringiem; w przypadku flagi ERR_TXT_MALLOCED, dane należy zwolnić za pomocą funkcji OPENSSL_free()

Opis tekstowy błędu można uzyskać za pomocą funkcji ERR_error_string() lub ERR_error_string_n().

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

char *ERR_error_string(unsigned long err, char *buf);
void ERR_error_string_n(unsigned long err, char *buf, size_t len);
```

Funkcja ERR_error_string() pobiera opis tekstowy reprezentujący kod błędu err i umieszcza go w buforze buf. Rozmiar bufora musi wynosić co najmniej 120 bajtów. Gdy buf jest równy NULL, opis błędu jest umieszczany w statycznym buforze. W aplikacjach wielowątkowych zaleca się stosowanie funkcji ERR_error_string_n(), która zapisuje co najwyżej len znaków (włączając zamykający string znak ‘\0’) do bufora buf. Parametr buf nie może być równy NULL.
Opis tekstowy błędu ma następujący format:

```
error:<kod błędu>:<nazwa biblioteki>:<nazwa funkcji>:<przyczyna błędu>
```

Przed użyciem funkcji ERR_error_string*(), opisy błędów muszą być załadowane do pamięci. W tym celu należy wywołać funkcję ERR_load_crypto_strings() lub SSL_load_error_strings():

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

void ERR_load_crypto_strings(void); /* Dla libcrypto. */

#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);
```

Dla wygody programistów, *OpenSSL* definiuje funkcję `ERR_print_errors_fp()`, która wysyła informacje na temat wszystkich błędów z kolejki wątku do strumienia `fp`.

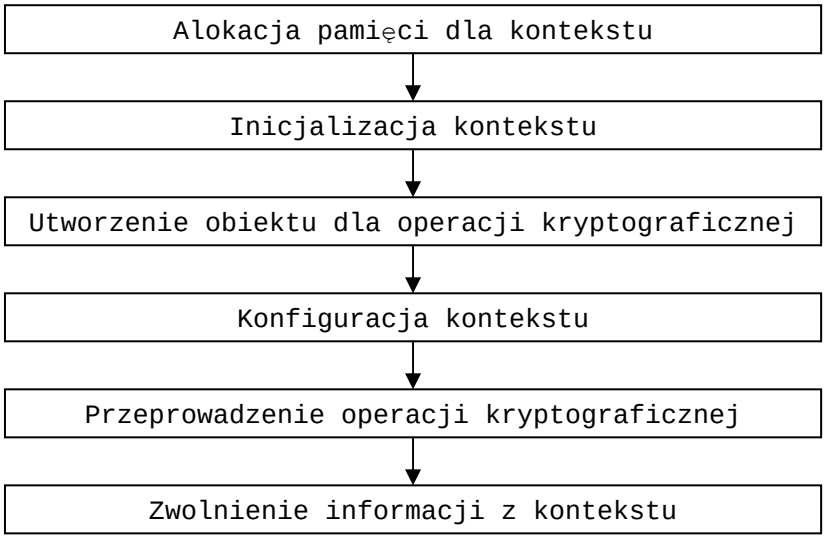
```
#include <openssl/err.h>

void ERR_print_errors_fp(FILE *fp);
```

Wywołanie funkcji `ERR_print_errors_fp()` powoduje opróżnienie kolejki błędów.

1.3.2. Interfejs EVP

Interfejs biblioteki *OpenSSL* jest bardzo rozległy. W zasadzie każdy algorytm szyfrujący oraz każda kryptograficzna funkcja skrótu posiada własny zestaw metod. *OpenSSL* dostarcza również API o nazwie EVP (ang. *Envelope Application Programming Interface*), które stanowi spójny i uogólniony interfejs dla większości operacji kryptograficznych.
Zasada korzystania z interfejsu EVP przedstawiona jest na poniższym rysunku.



Rys. 1. Zasada korzystania z interfejsu EVP.

Etapy przedstawione na rys. 1 zostaną omówione na przykładzie programu, który szyfruje dane algorytmem symetrycznym AES:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/err.h>
#include <openssl/evp.h>

int main(int argc, char **argv) {

int              retval, tmp, i;
char             plaintext[80] = “Laboratorium PUS.”;
unsigned char    ciphertext[80];
int              plaintext_len, ciphertext_len;

unsigned char    key[] = {0x00, 0x01, 0x02, /* (...) */, 0x31 };
unsigned char    iv[]  = {0x15, 0x14, 0x13, /* (...) */, 0x00 }

EVP_CIPHER_CTX *ctx = (EVP_CIPHER_CTX*)malloc(sizeof(EVP_CIPHER_CTX));

EVP_CIPHER_CTX_init(ctx);

const EVP_CIPHER *cipher  = EVP_aes_256_cbc();

retval = EVP_EncryptInit_ex(ctx, cipher, NULL, key, iv);
if (!retval) { /* Obsługa błędów */}

retval = EVP_EncryptUpdate(ctx, ciphertext, &ciphertext_len,
                           (unsigned char*)plaintext,
                           strlen(plaintext));
if (!retval) { /* Obsługa błędów */}

retval = EVP_EncryptFinal_ex(ctx, ciphertext + ciphertext_len, &tmp);
if (!retval) { /* Obsługa błędów */}

ciphertext_len += tmp;
EVP_CIPHER_CTX_cleanup(ctx); free(ctx);

fprintf(stdout, “Ciphertext (hex):\n”);
for (i = 0; i < ciphertext_len; i++) {
    fprintf(stdout, “%02x”, ciphertext[i]
}
exit(EXIT_SUCCESS);
}
```


Alokacja pamięci dla kontekstu może odbywać się statycznie lub dynamicznie. W omawianym przykładzie wykorzystano funkcję `malloc()` do przydzielenia obszaru pamięci dla struktury `EVP_CIPHER_CTX`.

```
EVP_CIPHER_CTX *ctx = (EVP_CIPHER_CTX*)malloc(sizeof(EVP_CIPHER_CTX));
```

Kolejnym krokiem jest **inicjalizacja kontekstu**:

```
EVP_CIPHER_CTX_init(ctx);
```

Konfiguracja kontekstu wymaga wcześniejszego **utworzenia obiektu dla operacji kryptograficznej**. Metoda `EVP_aes_256_cbc()` zwraca wskaźnik na obiekt reprezentujący algorytm AES w trybie CBC (ang. *Cipher Block Chaining*), który będzie wykorzystywany do szyfrowania danych:

```
const EVP_CIPHER *cipher  = EVP_aes_256_cbc();
```

Funkcja `EVP_EncryptInit_ex()` **konfiguruje kontekst** `ctx`, dla operacji szyfrowania algorytmem `cipher`.

```
retval = EVP_EncryptInit_ex(ctx, cipher, NULL, key, iv);
```

Argument `key` jest w omawianym przykładzie 256-bitowym kluczem dla algorytmu AES. Algorytm wykorzystuje tryb CBC – na bloku tekstu jawnego i poprzednim bloku zaszyfrowanym wykonywana jest operacja XOR. Dzięki temu wzorce zawarte w tekście jawnym nie są przenoszone na szyfrogram. Tryb CBC wymaga zastosowania wektora inicjalizacyjnego (na wektorze oraz pierwszym bloku tekstu jawnego zostanie wykonana operacja XOR). Symetryczne algorytmy blokowe (w tym AES) operują na blokach danych określonego rozmiaru. Wektor inicjalizacyjny (`iv`) powinien mieć rozmiar równy długości bloku, na którym operuje algorytm. Dla algorytmu AES jest to 128 bitów (16 bajtów).
Funkcja `EVP_EncryptInit_ex()` pozwala dodatkowo określić silnik kryptograficzny akceleratora sprzętowego. W przytoczonym przykładzie, parametr definiujący silnik kryptograficzny ma wartość `NULL`. Oznacza to, że zostanie użyta domyślna implementacja softwareowa.

Proces szyfrowania danych jest przeprowadzany za pomocą funkcji `EVP_EncryptUpdate()` i `EVP_EncryptFinal_ex()`.

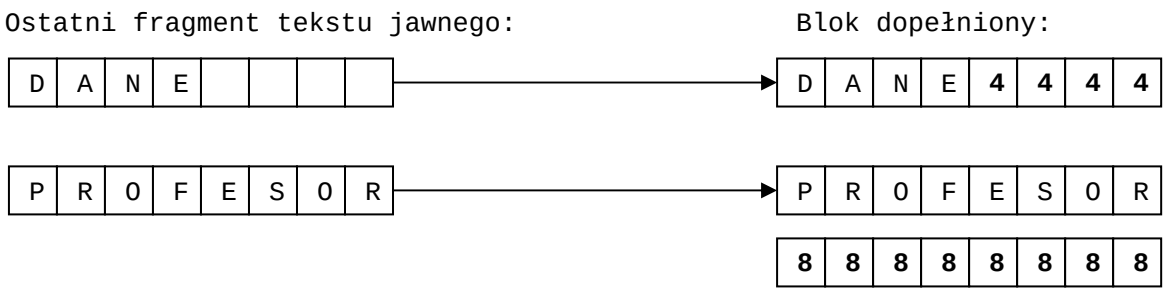

```
#include <openssl/evp.h>

int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx,
                      unsigned char *out,
                      int *outl,
                      unsigned char *in,
                      int inl);

int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx,
                        unsigned char *out,
                        int *outl);
```

Funkcja `EVP_EncryptUpdate()` szyfruje `inl` bajtów z bufora `in` i zapisuje szyfrogram do tablicy `out`. Może być wywoływana wiele razy w celu szyfrowania kolejnych bloków danych. Liczba bajtów zapisanych do tablicy `out` jest umieszczana w zmiennej o adresie `outl`.

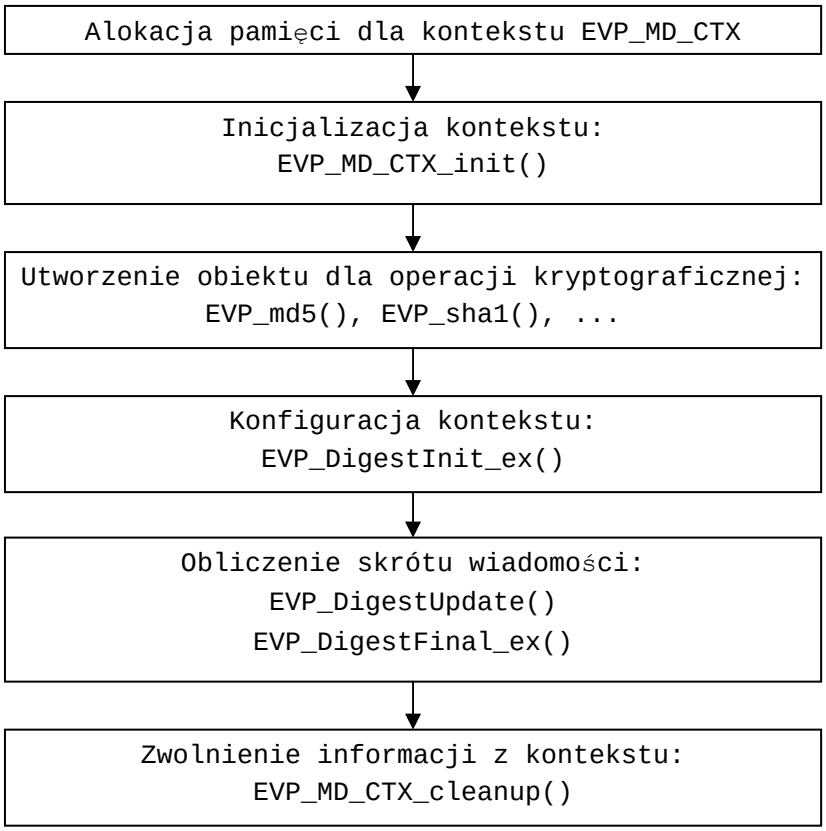
Funkcja `EVP_EncryptFinal_ex()` jest wykorzystywana do zakończenia procesu szyfrowania tekstu jawnego. Ostatnia część zaszyfrowanych danych jest umieszczana w buforze `out`, a liczba zapisanych bajtów jest zwracana za pomocą `outl`. W przypadku, gdy ostatni fragment tekstu jawnego ma rozmiar mniejszy od rozmiaru bloku, na którym operuje algorytm szyfrujący, konieczne jest zastosowanie dopełnienia. *OpenSSL* wykorzystuje domyślnie dopełnienie PKCS (ang. *Public Key Cryptography Standatds*) opracowane przez firmę *RSA Security*. Jeżeli szyfrowany fragment tekstu jawnego jest o N bajtów mniejszy od długości bloku w stosowanym algorytmie, to przed szyfrowaniem, funkcja `EVP_EncryptFinal_ex()` dopisuje do tego fragmentu N bajtów o wartości N. Podczas deszyfrowania sprawdzana jest wartość ostatniego bajta ostatniego bloku odszyfrowanej wiadomości i odrzucane jest tyle bajtów końcowych, ile ta wartość wskazuje. **Dopełnienie PKCS jest dodawane zawsze** – nawet jeśli ostatni fragment tekstu jawnego ma długość równą rozmiarowi bloku algorytmu szyfrującego (rys. 2).



Rys.2. Dopełnienie PKCS dla szyfru blokowego.

Po wykonaniu operacji kryptograficznych należy **zwolnić z pamięci kontekstu ważne informacje** (np. klucz algorytmu). Do tego celu została wykorzystana funkcja `EVP_CIPHER_CTX_cleanup()`.

Rys. 3 przedstawia kolejne etapy obliczenia kryptograficznego skrótu wiadomości za pomocą interfejsu EVP.



Rys. 3. Etapy obliczania kryptograficznego skrótu wiadomości.

Istnieją dwie metody tworzenia obiektów kryptograficznych. Pierwsza opiera się na funkcjach `EVP_md5()`, `EVP_aes_256_cbc()`, itd. W tym przypadku, każdemu algorytmowi lub funkcji skrótu odpowiada funkcja EVP.
Druga metoda umożliwia tworzenie obiektów kryptograficznych na podstawie stringu:

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

const EVP_MD *EVP_get_digestbyname(const char *name);
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
```

Przed wywołaniem powyższych funkcji należy dodać do wewnętrznej tablicy *OpenSSL* nazwy szyfrów lub funkcji skrótu. W tym celu można wykorzystać funkcje `OpenSSL_add_all*()`:

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

```



```
void OpenSSL_add_all_ciphers(void); /* Wczytanie nazw szyfrów */
void OpenSSL_add_all_digests(void); /* Wczytanie nazw funkcji skrótu */

/* Wczytanie nazw szyfrów i funkcji skrótu: */
void OpenSSL_add_all_algorithms(void);
```

Przykład wykorzystania funkcji `EVP_get_digestbyname()` przedstawiony jest poniżej:

```
OpenSSL_add_all_digests();

/*
 * Utworzenie obiektu dla kryptograficznej funkcji skrótu (na podstawie
 * nazwy określonej przez argument wywołania programu):
 */
const EVP_MD *md = EVP_get_digestbyname(argv[1]);
if (md == NULL) {
    fprintf(stderr, “Unknown message digest: %s\n“, argv[1]);
    exit(EXIT_FAILURE);
}

/* ( . . . ) */

EVP_cleanup();
```

Proszę zwrócić uwagę na wywołanie funkcji `EVP_cleanup()`, która zwalnia nazwy z pamięci, gdy nie są już potrzebne.

1.3.3. BIO

BIO (ang. *Basic Input/Output*) jest niskopoziomowym interfejsem, który dostarcza warstwę abstrakcji dla operacji I/O. Interfejs ukrywa wiele szczegółów implementacyjnych dotyczących operacji na danych.
Istnieją dwa typy obiektów (abstrakcji) BIO:

- *source*/*sink* – reprezentuje wejście lub wyjście danych, np.: plik, gniazdo sieciowe,
- *filter* – obiekt wykorzystywany do przetwarzania danych.

Obiekty (abstrakcje) BIO można łączyć ze sobą w określonej kolejności – tworzą wówczas łańcuch. Dane zapisywane do pierwszego elementu łańcucha, są przetwarzane w ustalonej kolejności przez pozostałe elementy. Podobnie jest z danymi odczytywanymi z łańcucha – przechodzą przez wszystkie jego elementy i poddawane są określonym transformacjom.

Obiekt niepowiązany z innymi jest traktowany jako łańcuch o jednym komponencie. Biblioteka *OpenSSL* dostarcza funkcji do tworzenia i usuwania abstrakcji BIO, łączenia abstrakcji w łańcuchy oraz odczytywania i zapisywania danych z łańcuchów.
Poniżej przedstawiony jest przykład szyfrowania danych za pomocą interfejsu BIO, który funkcjonalnie nie różni się od przytoczonego w podrozdziale 1.3.2 szyfrowania za pomocą interfejsu EVP.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>

int main(int argc, char **argv) {

char             plaintext[80] = “Laboratorium PUS.”;
int              plaintext_len = strlen(plaintext);
BUF_MEM          *bptr;
Int              i;

unsigned char    key[] = {0x00, 0x01, 0x02, /* (...) */, 0x31 };
unsigned char    iv[]  = {0x15, 0x14, 0x13, /* (...) */, 0x00 }

BIO *bio_encrypt, *bio_mem;

bio_encrypt = BIO_new(BIO_f_cipher());

BIO_set_cipher(bio_encrypt, EVP_aes_256_cbc(), key, iv, 1);

bio_mem = BIO_new(BIO_s_mem());

BIO_push(bio_encrypt, bio_mem);

if (BIO_write(bio_encrypt, plaintext, plaintext_len) <= 0) {
    /* Obsługa błędów */
}

if (BIO_flush(bio_encrypt) != -1) { /* Obsługa błędów */ }

BIO_get_mem_ptr(bio_mem, &bptr);
BIO_set_close(bio_mem, BIO_NOCLOSE);
BIO_free_all(bio_encrypt);

fprintf(stdout, “Ciphertext (hex):\n”);
```



```
for (i = 0; i < bptr->length; i++) {
    fprintf(stdout, “%02x”, (unsigned char)bptr->data[i]);
}
fprintf(stdout, “\n”);

BUF_MEM_free(bptr);

exit(EXIT_SUCCESS);
}
```

Funkcja `BIO_new()` przyjmuje wskaźnik na strukturę `BIO_METHOD`, która określa typ tworzonego obiektu:

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

BIO* BIO_new(BIO_METHOD* type);
```

OpenSSL implementuje szereg funkcji, które mogą zostać użyte w wywołaniu `BIO_new()`. W omawianym przykładzie, `BIO_s_mem()` zwraca wskaźnik na strukturę reprezentującą bufor w pamięci, a `BIO_f_cipher()` – wskaźnik na strukturę dla obiektu szyfrującego. Zapis „_s_” w nazwie metody oznacza, że zwraca ona obiekt typu *source*/*sink*. Z kolei zapis „_f_” oznacza, że zwracany jest obiekt filtrujący (przetwarzający dane).
Niektóre obiekty BIO mogą zostać użyte tuż po utworzeniu, część wymaga dodatkowej inicjalizacji. Dobrym przykładem jest tu obiekt BIO używany do szyfrowania danych:

```
bio_encrypt = BIO_new(BIO_f_cipher());
BIO_set_cipher(bio_encrypt, EVP_aes_256_cbc(), key, iv, 1);
```

Inicjalizacja obiektu szyfrującego została przeprowadzona za pomocą funkcji `BIO_set_cipher()`:

```
#include <openssl/bio.h>
#include <openssl/evp.h>

void BIO_set_cipher(BIO *b, const EVP_CIPHER *cipher,
                    unsigned char *key, unsigned char* iv, int enc);
```

Kolejne parametry funkcji to: konfigurowany obiekt BIO, algorytm szyfrujący, klucz, wektor inicjalizujący oraz tryb operacji (1 dla szyfrowania i 0 dla odszyfrowywania).

Po utworzeniu i opcjonalnej konfiguracji, obiekty BIO są najczęściej łączone w łańcuchy za pomocą funkcji `BIO_push()`.

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

BIO* BIO_push(BIO *b, BIO *append);
```

`BIO_push()` łączy łańcuch `b` z łańcuchem `append`. Proszę uwzględnić fakt, że obiekty BIO niezwiązane z innymi są traktowane jako jednoelementowe łańcuchy. Wywołanie:

```
BIO_push(bio_encrypt, bio_mem);
```

wiąże obiekt `bio_encrypt` z obiektem `bio_mem`. Zapis tekstu jawnego do obiektu `bio_encrypt` spowoduje zaszyfrowanie tekstu i umieszczenie szyfrogramu w buforze pamięci:

```
BIO_write(bio_encrypt, plaintext, plaintext_len);
```

W celu zwolnienia obiektu BIO z pamięci należy wywołać funkcję `BIO_free()`. Usunięcie wszystkich obiektów z łańcucha jest możliwe za pomocą `BIO_free_all()` – jako argument należy podać wskaźnik na pierwszy element łańcucha:

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

int BIO_free(BIO *a);
void BIO_free_all(BIO *a);
```

Usunięcie obiektu BIO może spowodować zwolnienie zasobów wykorzystywanych przez obiekt (pliku, gniazda sieciowego, bufora danych – w zależności od typu BIO). Zachowanie obiektu BIO w przypadku jego usunięcia definiuje funkcja `BIO_set_close()`:

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

BIO_set_close(BIO *b, long flag);
```

Parametr `flag` może przyjmować wartość `BIO_CLOSE` lub `BIO_NOCLOSE`. Jeżeli ustawiona jest flaga `BIO_CLOSE`, wszystkie zasoby wykorzystywane przez obiekt BIO zostaną poprawnie zamknięte lub zwolnienie podczas usuwania obiektu.

Przedstawiony poniżej fragment kodu źródłowego jest odpowiedzialny za usunięcie łańcucha obiektów BIO oraz zachowanie szyfrogramu w buforze pamięci:

```
BIO_get_mem_ptr(bio_mem, &bptr);
BIO_set_close(bio_mem, BIO_NOCLOSE);
BIO_free_all(bio_encrypt);
```

Wywołanie funkcji `BIO_get_mem_ptr()` pobiera wskaźnik na bufor związany z obiektem `bio_mem`. Struktura reprezentująca bufor jest przedstawiona poniżej:

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

typedef struct buf_mem_st {
         int length; /* Rozmiar danych w buforze w bajtach. */
         char *data; /* Wskaznik na dane. */
         int max; /* Rozmiar zaalokowanej pamięci. */
} BUF_MEM;
```

Obiekt `bio_mem` zarządza rozmiarem bufora dynamicznie (bufor jest powiększany w razie potrzeby, a jedynym limitem jest ilość dostępnej pamięci).
Usunięcie łańcucha BIO (w tym obiektu `bio_mem`) nie spowoduje usunięcia struktury `BUF_MEM` z pamięci, ponieważ ustawiona jest flaga `BIO_NOCLOSE`. Zwolnienie bufora `BUF_MEM` odbywa się niezależnie od obiektu BIO, za pomocą funkcji `BUF_MEM_free()`. Więcej informacji na temat buforowania można uzyskać na stronach podręcznika systemowego „*buffer*” (3ssl).

1.3.4. HMAC

Kody MAC (ang. *Message Authentication Code*) są wykorzystywane do uwierzytelniania źródła wiadomości i sprawdzania integralności danych. *OpenSSL* dostarcza implementację MAC bazującą na jednokierunkowych funkcjach skrótu – HMAC (ang. *Keyed Hashing MAC*). HMAC posiada własny interfejs, w skład którego wchodzą następujące funkcje:

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

HMAC_CTX_init(HMAC_CTX *ctx);

HMAC_Init_ex(HMAC_CTX *ctx, const void *key, int key_len,
             const EVP_MD *md);
```



```
HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, int len);

HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len);

HMAC_CTX_cleanup(HMAC_CTX *ctx);
```

Funkcja `HMAC_CTX_init()` inicjalizuje kontekst dla operacji obliczania MAC.
`HMAC_Init_ex()` przeprowadza konfigurację kontekstu – pozwala na określenie klucza (`key`), jego długości (`key_len`) oraz jednokierunkowej funkcji skrótu (`md`), która zostanie użyta do obliczenia MAC.
Po zainicjalizowaniu i konfiguracji, kontekst może zostać wykorzystany przez funkcję `HMAC_Update()`, która przyjmuje wskaźnik na bufor przechowujący wiadomość (`data`) oraz rozmiar wiadomości w bajtach (`len`). Wiadomość może zostać przekazana do funkcji `HMAC_Update()` w całości lub we fragmentach. Informacje wymagane do obliczenia MAC są przechowywane w kontekście między kolejnymi wywołaniami `HMAC_Update()`.
Funkcja `HMAC_Final()` finalizuje proces obliczania kodu uwierzytelniającego wiadomości. MAC jest umieszczany w buforze `md`, a rozmiar MAC w zmiennej wskazywanej przez `len`.
`HMAC_CTX_cleanup()` zwalnia informacje z kontekstu `ctx` (m.in. klucz użyty do utworzenia MAC).

1.3.5. RSA

Algorytm RSA został opublikowany przez Ronalda Rivesta, Adi Shamira i Leonarda Adlemana w 1977 roku. **RSA** jest akronimem utworzonym z pierwszych liter nazwisk jego twórców. Odtajnione w ostatnich latach dokumenty ujawniły, że metodę używaną w RSA opracował w 1973 roku Clifford Cocks z brytyjskiego GCHQ (ang. *Government Communications Headquarters*).
Podstawą bezpieczeństwa algorytmu jest trudność rozkładu dużych liczb na czynniki pierwsze (faktoryzacji). Klucze publiczny i prywatny są funkcjami pary dużych liczb pierwszych, a odtworzenie tekstu jawnego na podstawie szyfrogramu i klucza publicznego, którym został zaszyfrowany nie jest zadaniem łatwym.

Żeby wygenerować klucz RSA:

1. losujemy dwie duże liczby pierwsze p i q, oraz liczbę e względnie pierwszą z $(p-1)(q-1)$,
2. następnie obliczamy $d = e^{-1} \bmod (p-1)(q-1)$
   Ponieważ wybraliśmy względnie pierwsze e, ma ono odwrotność i obliczyć ją możemy rozszerzonym algorytmem Euklidesa.
3. obliczamy też $n = pq$.

Klucz publiczny to para *(e,n)*. Para *(d,n)* stanowi klucz prywatny. Liczbę *n* nazywa się modułem, *e* – wykładnikiem publicznym, a *d* – wykładnikiem prywatnym.
Dla danej wiadomości m zachodzi:

$c = m^e \bmod n$
$m = c^d \bmod n$

W celu zaszyfrowania wiadomości podnosimy jej liczbową reprezentację do potęgi *e* modulo *n*. Żeby odszyfrować wiadomość podnosimy *c* (szyfrogram) do potęgi *d* modulo *n*.

Nie znając *d* nie potrafimy łatwo odzyskać wiadomości z szyfrogramu.
Nie znając faktoryzacji *n* na *p* i *q,* nie znamy też prostej metody odtworzenia *d* z *e*.

Długość wygenerowanego klucza RSA jest określona długością modułu *n* (każda z liczb *p* i *q* musi mieć w przybliżeniu długość połowy klucza). Długość wykładnika publicznego *e* nie musi być jednak równa długości wykładnika prywatnego *d*. Często wykładnik publiczny jest mniejszy od prywatnego, co powoduje, że korzystanie z klucza prywatnego jest znacznie wolniejsze od korzystania z publicznego. Dzięki temu szyfrowanie danych kluczem publicznym jest szybkie i można je wykonać na klientach o ograniczonej mocy obliczeniowej. Dodatkową zaletą jest szybkie sprawdzanie podpisów cyfrowych (do weryfikacji podpisu jest wykorzystywany klucz publiczny). Dobrą praktyką jest więc taki dobór wykładnika publicznego, aby szyfrowanie było mało kosztowne obliczeniowe, po czym wykładnik prywatny jest generowany wg wzoru: $d = e^{-1} \bmod (p-1)(q-1)$.
Standard X.509 zaleca stosowanie wartości F_4 dla wykładnika publicznego. F_4 jest liczbą pierwszą Fermata, równą 65537 (0x10001). Ogólny wzór na wyprowadzanie liczb Fermata przedstawiony jest poniżej:

$$F_n = 2^{2^n} + 1$$

Do chwili obecnej, jedynymi znanymi liczbami pierwszymi Fermata są F_0, F_1, F_2, F_3, F_4 i nie wiadomo, czy jest ich więcej.

Interfejs *OpenSSL* umożliwia generowanie kluczy oraz przeprowadzanie operacji kryptograficznych za pomocą algorytmu RSA (szyfrowanie, deszyfrowanie danych, tworzenie i weryfikacji podpisów cyfrowych). Warto nadmienić, że proces szyfrowania algorytmem RSA jest bardziej złożony obliczeniowo od szyfrowania za pomocą algorytmów symetrycznych. Z tego względu RSA wykorzystuje się głównie do tworzenia podpisów cyfrowych i szyfrowania małych ilości danych (np. podczas uzgadniania tajnego klucza między dwoma stronami). Korzystanie z RSA podczas szyfrowania danych wiąże się z ograniczeniem rozmiaru szyfrowanej wiadomości. Wiadomość w reprezen-

tacji liczbowej powinna mieć długość mniejszą od długości stosowanego klucza. Dodatkowo, rozmiar wiadomości może być ograniczony przez dopełnienie. W przypadku algorytmu RSA dopełnienie (najczęściej PKCS lub OAEP) dodawane jest przed liczbową reprezentacją wiadomości i może zajmować kilkanaście bajtów (11 dla PKCS).

Za generowanie pary kluczy odpowiedzialna jest funkcja `RSA_generate_key()`:

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

RSA *RSA_generate_key(int num, unsigned long e,
                 void (*callback)(int,int,void*),
                 void *cb_arg);
```

Parametr	Opis
`num`	rozmiar modułu *n* (n = pq)
`e`	wartość publicznego wykładnika *e*; liczba Fermata F_4 zdefiniowana jest jako `RSA_F4`
`callback`	wskaźnik na funkcję, która raportuje postęp generowania kluczy; może wynosić `NULL`
`cb_arg`	Dane przekazywane do funkcji `callback`

Funkcja zwraca wskaźnik na strukturę przechowującą parę kluczy lub wartość `NULL` w przypadku błędu. Struktura jest alokowana dynamicznie i musi być zwolniona za pomocą `RSA_free()`:

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

void RSA_free(RSA *rsa);
```

Przed wywołaniem funkcji `RSA_generate_key()` należy zainicjalizować generator liczb pseudolosowych. Jedną z metod stosowanych w systemie *Linux* jest wykorzystanie pliku `/dev/urandom` jako źródła entropii:

```
/*
 * Inicjalizacja generatora liczb pseudolosowych za pomocą
 * 128 bajtów z pliku /dev/urandom:
 */
RAND_load_file(“/dev/urandom”, 128);
```

Często zachodzi konieczność wymiany klucza publicznego przez sieć lub zapisu kluczy do pliku. Informacje przechowywane przez strukturę RSA muszą być wówczas

odpowiednio sformatowane i zakodowane. Istnieją dwie podstawowe metody kodowania: DER (ang. *Distinguished Encoding Rules*) i PEM (ang. *Privacy Enhanced Mail*).
DER jest formatem binarnym, co świetnie sprawdza się w przypadku przechowywania informacji w pamięci lub przesyłania danych przez sieć. Format PEM (opisany w RFC 1421-1424) polega na zakodowaniu w formacie Base 64 kodu DER oraz dodaniu nagłówka i stopki ASCII. Format PEM jest stosowany w przypadku komunikacji tekstowej, np. przez e-mail.
OpenSSL udostępnia kilkadziesiąt funkcji obsługujących formaty DER i PEM, w tym funkcje umożliwiające zakodowanie kluczy RSA.
Poniżej przedstawiony jest przykład wygenerowania kluczy RSA i zapisu klucza prywatnego (w formacie PEM) do pliku. Klucz prywatny przed zakodowaniem za pomocą Base 64 jest szyfrowany. Użytkownik jest pytany o hasło, na podstawie którego zostanie wygenerowany klucz dla algorytmu AES.

```
int retval;

/* Wskaznik na plik: */
FILE *file;

/* Wskaznik na strukture przechowujaca pare kluczy RSA: */
RSA *key_pair;

RAND_load_file("/dev/urandom", 128);

/* Wygenerowanie kluczy: */
key_pair = RSA_generate_key(1024, RSA_F4, NULL, NULL);
if (key_pair == NULL) { /* Obsługa błędu */ }

file = fopen("private.key", "w");
if (file == NULL) { /* Obsługa błędu */ }

retval = PEM_write_RSAPrivateKey(file, key_pair, EVP_aes_256_cbc(),
                                 NULL, 0, NULL, NULL);
if (!retval) { /* Obsługa błędu */ }

fclose(file);

/* Zwolnienie pamieci: */
RSA_free(key_pair);
```

Funkcja `PEM_write_RSAPrivateKey()` jest odpowiedzialna za zapis klucza prywatnego do pliku:


```
#include <openssl/pem.h>
PEM_write_RSAPrivateKey(FILE *fp, RSA *x, const EVP_CIPHER *enc,
                        unsigned char *kstr, int klen,
                        pem_password_cb *cb, void *u);
```

Parametr	Opis
fp	wskaźnik na strukturę FILE (plik do zapisu)
x	wskaźnik na strukturę RSA przechowującą parę kluczy
enc	algorytm szyfrujący; jeżeli enc ma wartość NULL, to klucz zostanie zapisany do pliku bez szyfrowania
cb	funkcja *callback*, której zadaniem jest zapytanie użytkownika o hasło (na podstawie hasła zostanie wygenerowany klucz dla algorytmu symetrycznego);
kstr	jeżeli kstr jest różne od NULL, to klen bajtów wskazywanych przez kstr zostanie użyte jako hasło
klen	liczba bajtów w buforze kstr, które mają zostać wykorzystane jako hasło (jeżeli kstr jest różne od NULL)
u	jeżeli parametr cb jest równy NULL i u jest różne od NULL, to u wskazuje na string (ciąg znaków zakończony ‘\0’), który zostanie użyty jako hasło; jeżeli cb i u są równe NULL, zostanie wykorzystana domyśla funkcja *callback* pytająca o hasło na terminalu

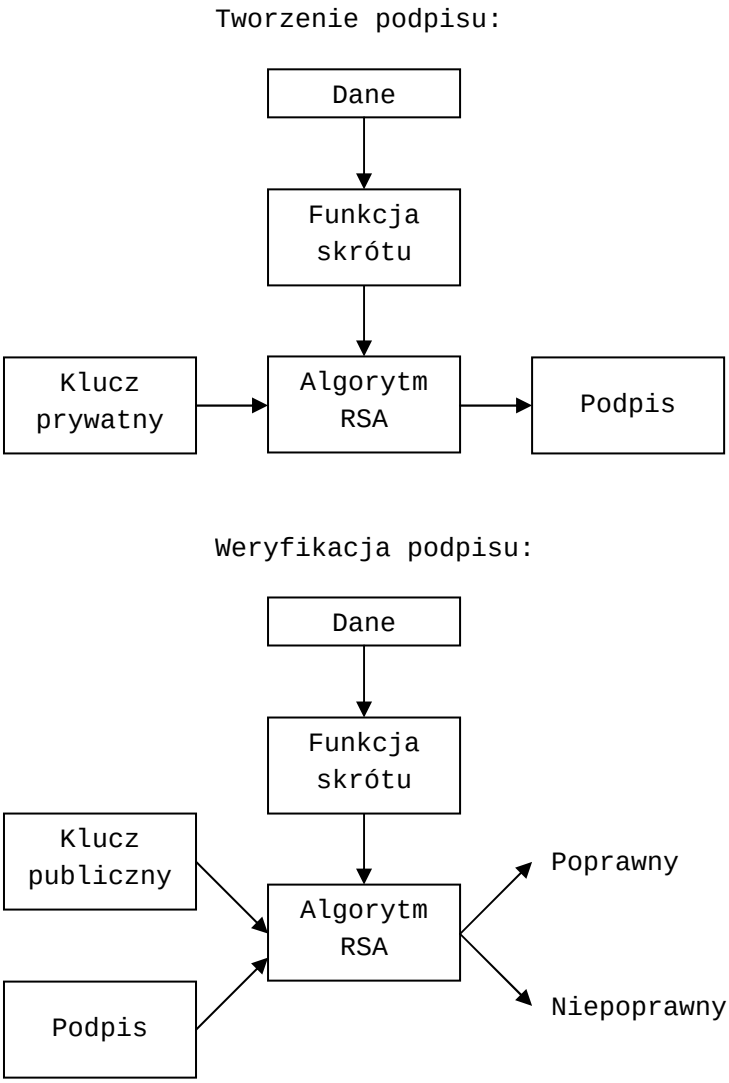
Informacje na temat pozostałych funkcji dotyczących kodowania PEM można znaleźć na stronie podręcznika systemowego „*pem*” (3ssl). Proszę zapoznać się z zastosowaniem następujących funkcji:

- PEM_write_RSAPublicKey(),
- PEM_read_RSAPublicKey(),
- PEM_read_RSAPrivateKey().

Jednym z głównych zastosowań algorytmu RSA są podpisy cyfrowe. Podpis cyfrowy jest tworzony na podstawie tajnych danych znanych wyłącznie jego autorowi, ale może zostać sprawdzony za pomocą informacji publicznie udostępnianych przez podpisującego. Rolę informacji tajnych i publicznie dostępnych pełnią odpowiednio klucz prywatny i publiczny. Podpis cyfrowy nie tylko poświadcza autentyczność wiadomości, ale również zapewnia integralność danych i czyni wiadomość niezaprzeczalną. Jeżeli użytkownik zaprzeczy, że wysłał podpisaną cyfrowo wiadomość, to istnieją tylko dwie możliwości:

a) klucz prywatny użytkownika został skradziony,
b) użytkownik mija się z prawdą.

Z podpisami cyfrowymi wiążą się dwa procesy: tworzenie podpisu i sprawdzanie podpisu (rys. 4) [12].



Rys. 4. Tworzenie i weryfikacja podpisu cyfrowego za pomocą algorytmu RSA.

Operacje tworzenia i weryfikacji podpisu cyfrowego mogą zostać przeprowadzone za pomocą funkcji `RSA_sign()` i `RSA_verify()`:

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

int RSA_sign(int type, const unsigned char *m, unsigned int m_len,
             unsigned char *sigret, unsigned int *siglen, RSA* rsa);

int RSA_verify(int type, const unsigned char *m, unsigned int m_len,
             unsigned char *sigbuf, unsigned int siglen, RSA* rsa);
```


`RSA_sign()` podpisuje wiadomość `m` o rozmiarze `m_len` wykorzystując klucz prywatny `rsa`. Sygnatura jest zapisywana do `sigret`, a jej rozmiar do `siglen`. Bufor `sigret` powinien mieć rozmiar nie mniejszy od `RSA_size(rsa)`. Parametr `type` jest identyfikatorem NID (ang. *Numerical identifier*) określającym jednokierunkową funkcję skrótu, która ma zostać wykorzystana do utworzenia podpisu cyfrowego. Lista identyfikatorów znajduje się w pliku `<openssl/objects.h>`. Najczęściej stosowane identyfikatory to:

- `NID_sha1`,
- `NID_ripemd160`,
- `NID_md5`.

W przypadku powodzenia, funkcja `RSA_sign()` zwraca wartość 1.

Funkcja `RSA_verify()` generuje podpis cyfrowy dla wiadomości `m` i sprawdza, czy odpowiada on podpisowi w buforze `sigbuf`. Rozmiar weryfikowanego podpisu jest określony przez parametr `siglen`, natomiast rozmiar wiadomości przez `m_len`.
Funkcja `RSA_verify()` wymaga podania:

- identyfikatora funkcji skrótu (`type`), która została wykorzystana do utworzenia podpisu w buforze `sigbuf`,
- klucza publicznego podmiotu (`rsa`), który podpisał wiadomość (utworzył podpis w buforze `sigbuf`).

W przypadku, gdy podpisy cyfrowe są zgodne, funkcja zwraca wartość 1.

1.4. Cel laboratorium

Celem laboratorium jest zapoznanie się z biblioteką kryptograficzną *OpenSSL* (instrukcja nie obejmuje zagadnień związanych z protokołami SSL/TLS). Po zrealizowaniu laboratorium powinieneś:

- umieć korzystać z szyfrów symetrycznych za pomocą interfejsu EVP oraz BIO,
- rozumieć podstawowe tryby szyfrowania oraz dopełnienia,
- rozumieć przeznaczenie skrótów wiadomości i kodów MAC oraz zasady ich tworzenia w *OpenSSL*,
- znać i rozumieć algorytm RSA,
- umieć tworzyć i weryfikować podpisy cyfrowe.

2. Przebieg laboratorium

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

2.1. Zadanie 1. Szyfrowanie w trybie ECB

Zadanie polega na analizie kodu przykładowego programu. Program demonstruje proces szyfrowania i deszyfrowania danych z wykorzystaniem interfejsu EVP. Szyfrowanie odbywa się za pomocą algorytmu AES w trybie ECB. Klucz (128 bitów) jest tablicą bajtów zdefiniowaną w kodzie programu - dzięki temu wynik szyfrowania jest przewidywalny (zawsze otrzymamy ten sam szyfrogram dla ustalonego ciągu wejściowego).
Podczas uruchamiania programu, należy podać wartość określającą, czy podczas szyfrowania/deszyfrowania danych ma zostać użyte dopełnienie PKCS (0 – nie, 1 – tak). Proszę zwrócić szczególną uwagę na parametry algorytmu szyfrującego (wypisywane za pomocą funkcji EVP) oraz na relacje między rozmiarem tekstu jawnego i szyfrogramu.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc –o cipher_ecb –lcrypto cipher_ecb.c`
3. Uruchomić program z różnymi wartościami argumentu, który określa, czy dopełnienie ma zostać użyte:
   `$ ./cipher_ecb <padding>`
   Jak zmienia się rozmiar szyfrogramu w zależności od wartości argumentu `<padding>`?
4. Czy między blokami tekstu jawnego, na których operuje algorytm, a blokami szyfrogramu występuje jakaś zależność?
5. Proszę zmodyfikować tekst jawny w taki sposób, aby rozmiar wiadomości nie był wielokrotnością rozmiaru bloku algorytmu (np.: dopisać na koniec wiadomości jedną literę). Proszę ponownie skompilować program i przeanalizować działanie programu dla operacji z dopełnieniem i bez dopełnienia.

2.2. Zadanie 2. Szyfrowanie w trybie CBC

Zadanie polega na analizie kodu przykładowego programu. Zasada działania programu jest analogiczna do opisanej w poprzednim zadaniu. Jedyną różnicą jest to, że program wykorzystuje algorytm AES w trybie CBC i wymaga zastosowania wektora inicjalizacyjnego.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc –o cipher_cbc –lcrypto cipher_cbc.c`
3. Uruchomić program z różnymi wartościami argumentu, który określa, czy dopełnienie ma zostać użyte:
   `$ ./cipher_cbc <padding>`
   Jak zmienia się rozmiar szyfrogramu w zależności od wartości argumentu `<padding>`?
4. Czy między blokami tekstu jawnego, na których operuje algorytm, a blokami szyfrogramu występuje jakaś zależność?
5. Proszę zmodyfikować tekst jawny w taki sposób, aby rozmiar wiadomości nie był wielokrotnością rozmiaru bloku algorytmu (np.: dopisać na koniec wiadomości jedną literę). Proszę ponownie skompilować program i przeanalizować działanie programu dla operacji z dopełnieniem i bez dopełnienia.

2.3. Zadanie 3. BIO

Celem zadania jest analiza programu szyfrującego dane za pomocą interfejsu BIO. Program pobiera dane ze standardowego wejścia, szyfruje je za pomocą algorytmu AES, koduje za pomocą Base 64 i wypisuje rezultat na standardowe wyjście. Wszystko odbywa się za pomocą łańcucha elementów BIO.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc –o cipher_bio –lcrypto cipher_bio.c`
3. Uruchomić program (dane można przekierować z pliku na standardowe wejście lub wprowadzić z klawiatury):
   `$ ./cipher_bio < plaintext`
4. Proszę porównać zakodowany za pomocą Base 64 szyfrogram z wynikiem polecenia:

```
openssl enc -aes-128-cbc -K 000102030405060708090000102030405 \
-iv 000102030405060708090000102030405 -a -in plaintext
```


2.4. Zadanie 4. Obliczanie skrótu wiadomości

Zadanie polega na analizie kodu przykładowego programu. Program przyjmuje jako argument wywołania nazwę jednokierunkowej funkcji mieszającej, która zostanie wykorzystana do obliczenia kryptograficznego skrótu wiadomości. Wiadomość można wprowadzić z klawiatury lub przekierować na standardowe wejście procesu z pliku. Podczas działa programu proszę zwrócić uwagę na parametry funkcji skrótu (rozmiar generowanego skrótu, rozmiar bloku na jakim operuje funkcja).

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:
   `$ gcc –o digest –lcrypto digest.c`
3. Uruchomić program podając nazwę funkcji skrótu („md5”, „sha1”, „ripemd160”, itp.):
   `$ ./digest <nazwa funkcji> < plaintext`
4. Proszę porównać skróty generowane przez program ze skrótami otrzymanymi za pomocą polecenia:
   `$ openssl <nazwa funkcji> plaintext`
   gdzie `<nazwa funkcji>`, to „md5”, „sha1”, „rmd160”, itp.

2.5. Zadanie 5. HMAC

Proszę zmodyfikować kod źródłowy z poprzedniego zadania w taki sposób, aby program zamiast kryptograficznego skrótu wiadomości tworzył kod uwierzytelniający (HMAC). Część kodu pozostanie bez zmian, m.in. fragment odpowiedzialny za utworzenie struktury `EVP_MD` dla funkcji skrótu oraz fragment odbierający wiadomość ze standardowego wejścia. Klucz wymagany przez algorytm MAC proszę zdefiniować jako tablicę bajtów (klucz można skopiować z zadania 1 lub 2).

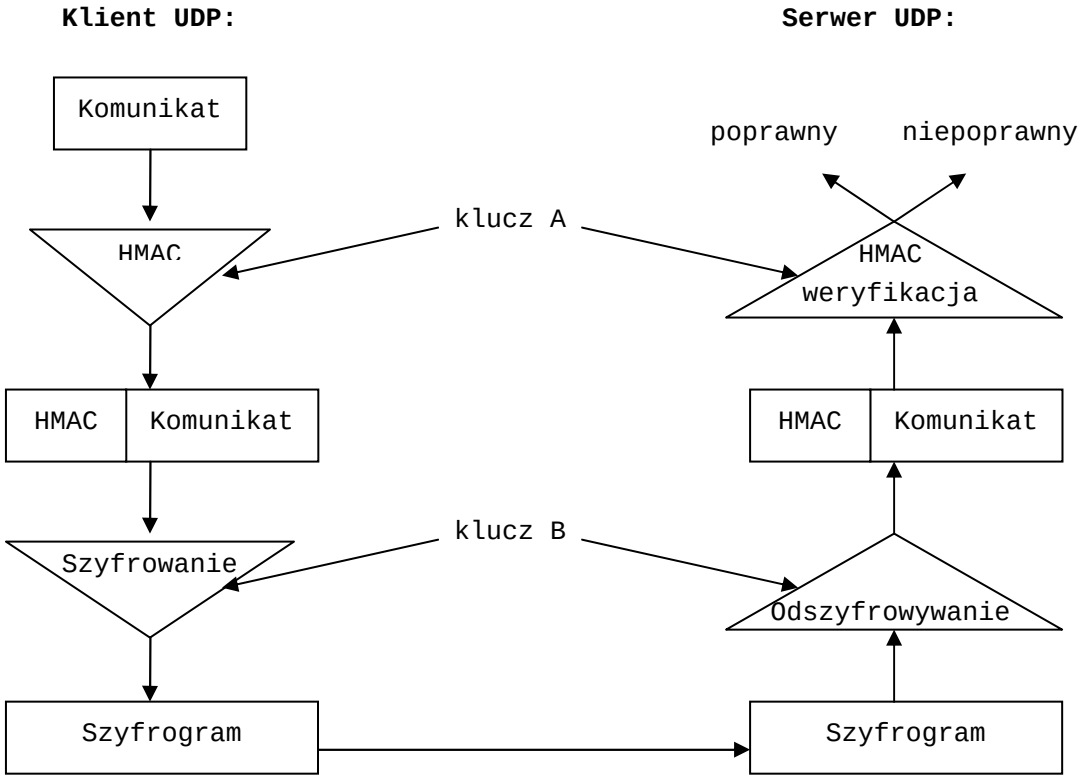
W celu uruchomienia programu należy wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:
   `$ gcc –o mac –lssl mac.c`
2. Uruchomić program (dane można przekierować z pliku na standardowe wejście lub wprowadzić z klawiatury):
   `$ ./mac < plaintext`

2.6. Zadanie 6. Praktyczne wykorzystanie szyfrowania i HMAC w komunikacji klient-serwer

Zadanie opiera się na programach klienta oraz serwera UDP. Klient przesyła wiadomość „Laboratorium PUS.” na adres i numer portu podane w argumentach wywołania. Serwer oczekuje na wiadomość na wybranym porcie. Po otrzymaniu data-

gramu UDP, serwer wypisuje komunikat na standardowe wyjście, po czym kończy działanie. Proszę zmodyfikować programy w taki sposób, aby zamiast tekstu jawnego przesyłany był szyfrogram. Dodatkowo proszę zabezpieczyć wiadomość kodem uwierzytelniającym (HMAC). Program serwera ma zweryfikować kod uwierzytelniający i wypisać odpowiedni komunikat na standardowe wyjście (rys.5). Do utworzenia HMAC należy wykorzystać funkcję MD5. W celu zwiększenia bezpieczeństwa, można zastosować niezależne klucze PSK (ang. *Pre-shared Key*) – dla szyfru symetrycznego i HMAC.
Ułatwieniem może być wykorzystanie kodu źródłowego programów z zadania 1 oraz z zadania 4.



Rys. 5. Zasada działania programów klienta i serwera UDP.

W celu uruchomienia programów należy wykonać następujące czynności:

1. Skompilować programy źródłowe do postaci binarnej:
   ```
   $ gcc –o client –lcrypto –lssl client.c
   $ gcc –o server –lcrypto –lssl server.c
   ```
2. Uruchomić sniffer (np. *tcpdump*) z opcją przechwytywania datagramów UDP.
3. Uruchomić program serwera na wybranym porcie:
   ```
   $ ./server <numer portu>
   ```
4. Uruchomić klienta UDP podając adres IP (np. 127.0.0.1) i numer portu serwera:
   ```
   $ ./client <adres IP> <numer portu>
   ```


5. Proszę upewnić się, że komunikat został poprawnie odszyfrowany i weryfikacja HMAC zakończyła się powodzeniem. Czy za pomocą sniffera da się odczytać treść komunikatu?

2.7. Zadanie 7. Generowanie i weryfikacja podpisu cyfrowego algorytmem RSA

Zadanie polega na analizie kodu przykładowych programów. Pierwszy program (*rsa*) generuje parę kluczy RSA i zapisuje klucze do plików *public.key* oraz *private.key* w bieżącym katalogu roboczym. Klucze są zapisywane w formacie PEM, a klucz prywatny jest dodatkowo szyfrowany algorytmem AES. Proszę zwrócić uwagę na fakt, że szyfrogram jest kodowany za pomocą Base 64 (wymaga tego standard PEM). Podczas zapisu klucza prywatnego, użytkownik zostanie zapytany o hasło, które posłuży do wygenerowania klucza dla algorytmu AES.
Dwa pozostałe programy: *client_rsa* oraz *server_rsa* są zmodyfikowaną wersją programów klienta i serwera UDP opisanych w zadaniu 5. Program klienta odczytuje klucz prywatny z pliku *private.key* i na jego podstawie generuje podpis cyfrowy dla wiadomości „Laboratorium PUS.”. Wiadomość wraz z podpisem jest przesyłana do serwera, który weryfikuje podpis za pomocą klucza publicznego z pliku *public.key*. Wynik weryfikacji jest wypisywany na standardowe wyjście.

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 –o rsa –lcrypto rsa.c
   $ gcc –o client_rsa –lcrypto client_rsa.c
   $ gcc –o server_rsa –lcrypto server_rsa.c
   ```
3. Wygenerować i zapisać do plików parę kluczy RSA:
   ```
   $ ./rsa
   ```
6. Uruchomić program serwera na wybranym porcie:
   ```
   $ ./server_rsa <numer portu>
   ```
7. Uruchomić klienta UDP podając adres IP (np. 127.0.0.1) i numer portu serwera:
   ```
   $ ./client_rsa <adres IP> <numer portu>
   ```


3. Opracowanie i sprawozdanie

Realizacja laboratorium pt. *„Kryptografia 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.

