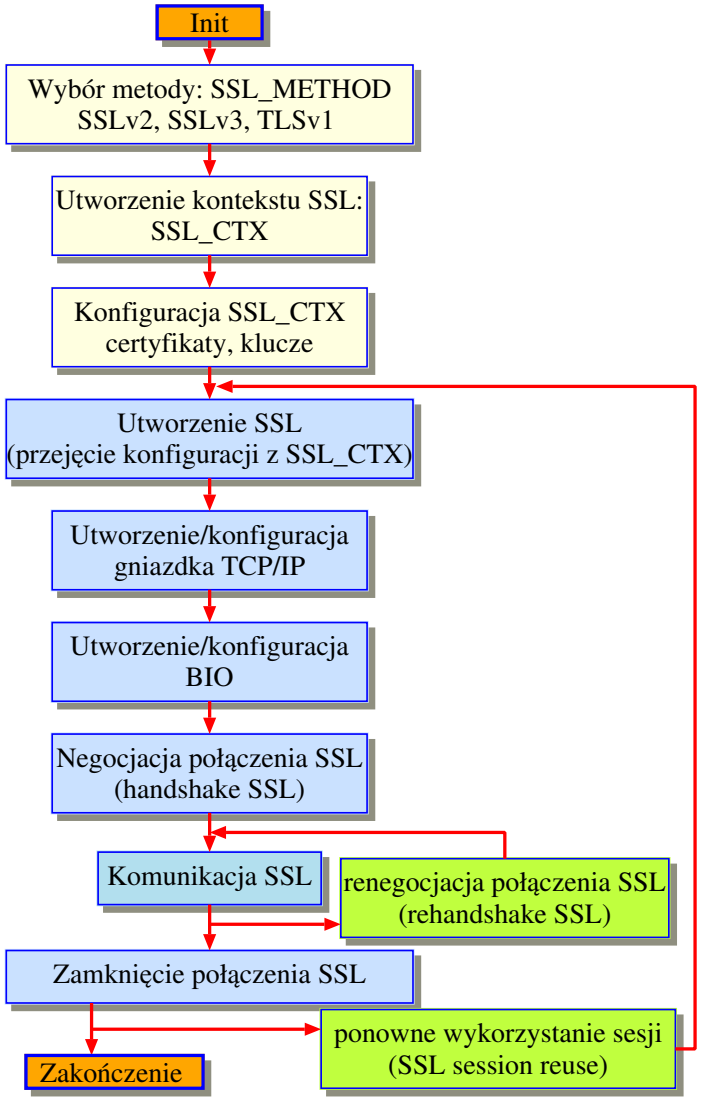
Bezpieczeństwo Usług Sieciowych
SSL API

dr inż. Tomasz Surmacz

28 listopada 2011

SSL – API

- Utworzenie kontekstu SSL (SSL_CTX)
  - do kontekstu można dodawać certyfikaty i klucze prywatne, a także wybierać dopuszczalne metody szyfrowania
  - w kontekście należy zdefiniować, jaka wersja SSL/TLS będzie wykorzystywana
- Niezależnie – należy nawiązać połączenie TCP/IP klient-serwer oraz utworzyć strukturę SSL zarządzającą sesją SSL.
- Istniejące połączenie i sesję SSL należy ze sobą powiązać poprzez BIO (buforowane wejście/wyjście) lub niejawnie
- Wymiana certyfikatów, ustalenie szyfrów, sprawdzenie kluczy, itp., następuje w momencie negocjacji połączenia SSL (funkcje *SSL_connect*()/*SSL_accept*())
- Bezpieczna komunikacja odbywa się następnie z użyciem struktury SSL (*SSL_write*()/*SSL_read*()) lub BIO (np. *BIO_puts*(), *BIO_write*() itp.)
- W trakcie sesji można renegocjować parametry połączenia – szyfry, klucze...
- Zamknięcie sesji: *SSL_shutdown*()



SSL API – setup

```
#include <openssl/ssl.h>
int SSL_library_init(void);   /* load encryption & hash algorithms for SSL */

#define OpenSSL_add_ssl_algorithms()    SSL_library_init()
#define SSLeay_add_ssl_algorithms()     SSL_library_init()

SSL_load_error_strings();     /* load the error strings and register the libssl error strings*/
```

- *SSL_library_init()* rejestruje wszystkie funkcje kryptograficzne i funkcje skrótu w SSL API.
- Standardowo ładowane algorytmy to DES-CBC, DES-EDE3-CBC, RC2 i RC4 oraz funkcje skrótu MD2, MD5 i SHA
- Funkcja *SSL_library_init()* zawsze zwraca wartość 1.
- *SSL_load_error_strings()* wczytuje i rejestruje w API komunikaty o błędach, które są używane przez funkcje należące do SSL API i Crypto API.

SSL API – wybór metody

- Wybór wersji protokołu SSL/TLS następuje przez wybór metody w strukturze SSL_METHOD

Typ protokołu	klient/serwer	dedykowany serwer	dedykowany klient
SSLv2	*SSLv2_method*(void)	*SSLv2_server_method*(void)	*SSLv2_client_method*(void)
SSLv3	*SSLv3_method*(void)	*SSLv3_server_method*(void)	*SSLv3_client_method*(void)
TLSv1	*TLSv1_method*(void)	*TLSv1_server_method*(void)	*TLSv1_client_method*(void)
SSLv23	*SSLv23_method*(void)	*SSLv23_server_method*(void)	*SSLv23_client_method*(void)

UWAGA! – pseudometoda SSLv23 pozwala wybrać SSLv2, SSLv3 lub TLSv1 zgodnie z oczekiwaniami drugiej strony.

  - serwer korzystający z SSLv23 zrozumie komunikaty HELLO dowolnej wersji protokołu i nawiąże połączenie z dowolnym klientem
  - klient korzystający z SSLv23 nie dogada się z serweremn SSLv3/TLSv1 ponieważ domyślnie wysyłany jest komunikat HELLO w wersji SSLv2.

- Struktura SSL_METHOD jest następnie używana do utworzenia struktury SSL_CTX

```
#include <openssl/ssl.h>
SSL_CTX *SSL_CTX_new(SSL_METHOD *method);
```

- Przykład kodu:

```
method = SSLv3_method();
ctx = SSL_CTX_new(method);
```

SSL API – wybór kluczy i certyfikatów

- Wybór certyfikatu (własnego) następuje poprzez wczytanie ich do struktury `SSL_CTX` za pomocą jednej z funkcji *SSL_CTX_use_certificate_file()* (z pliku), *SSL_CTX_use_certificate()* (ze struktury `X509`) lub *SSL_CTX_use_certificate_ASN1()* (z pamięci).
  - Serwer wczytuje:
    - Swój własny certyfikat (obowiązkowo)
    - Certyfikat CA (opcjonalnie)
  - Klient wczytuje:
    - Certyfikat CA (obowiązkowo)
    - Swój własny certyfikat (opcjonalnie)
- Wczytany certyfikat jest wysyłany do drugiej strony, która używa go do zaszyfrowania losowych danych
- Odszyfrowanie tych danych wymaga użycia klucza prywatnego związanego z wysłanym certyfikatem
- Wczytanie klucza prywatnego następuje za pomocą jednej z funkcji, odpowiedniej dla typu używanego certyfikatu

```
SSL_CTX_use_PrivateKey()                SSL_use_PrivateKey()
SSL_CTX_use_PrivateKey_ASN1()           SSL_use_PrivateKey_ASN1()
SSL_CTX_use_PrivateKey_file()           SSL_use_PrivateKey_file()
SSL_CTX_use_RSAPrivateKey()             SSL_use_RSAPrivateKey()
SSL_CTX_use_RSAPrivateKey_ASN1()        SSL_use_RSAPrivateKey_ASN1()
SSL_CTX_use_RSAPrivateKey_file()        SSL_use_RSAPrivateKey_file()
```

- Inne funkcje: *SSL_CTX_check_private_key()*, *SSL_CTX_use_certificate_chain_file()*.

SSL API – klucze i hasła

- Wczytane klucze mogą wymagać odblokowania hasłem
- W tym celu można zarejestrować funkcję callback wywoływaną w razie potrzeby

```
void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, pem_password_cb *cb);
void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx, void *u);

int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata);
```

- *SSL_CTX_set_default_passwd_cb_userdata*(ctx, userdata) ustawia wskaźnik na dane `userdata`, które będą przekazane w momencie wywołania funkcji *pem_passwd_cb*()
- Funkcja *pem_passwd_cb*() może zażądać interaktywnego podania hasła przez użytkownika albo pobrać je z pamięci (np. jeśli jest potrzebne do odblokowania kilku kluczy, warto spytać o nie tylko raz).
- Flaga `rwflag=1` oznacza, czy coś, do czego potrzebne jest hasło, będzie szyfrowane. W takim wypadku dobrą praktyką jest spytanie o hasło dwa razy i sprawdzenie zgodności, by wyeliminować pomyłki/literówki.
- Przykład: funkcja zwracająca hasło dostarczane przez `userdata`:

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *password)
{
    strncpy(buf, (char*)(password), size);
    buf[size-1] = '\0';
    return(strlen(buf));
}
```

SSL API – wybór kluczy i certyfikatów: CA

- Wczytanie certyfikatu CA

```
#include <openssl/ssl.h>
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile, const char *CApath);
```

- Ustala plik oraz ścieżkę do plików zawierających certyfikaty CA. Wymagane certyfikaty są w pierwszym rzędzie szukane w *CAfile*, potem w innych plikach w katalogu *CApath*. Jeden z parametrów funkcji może być ustawiony na `NULL`
- Plik wskazywany przez *CAfile* powinien zawierać jeden lub wiele certyfikatów w postaci BASE64 (postać „PEM"), tzn. między liniami postaci:

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

- Pliki w katalogu *CApath* wyszukiwanie są wg skrótu nazwy (np. *9d66eef0.0*), bez względu na rozszerzenie. Pliki/linki o odpowiednich nazwach można utworzyć za pomocą programu `c_rehash`.
- Serwer żądający certyfikatu klienta musi wcześniej wysłać listę CA, które są akceptowalne w celu weryfikacji certyfikatu klienta. Lista ta jest niezależna od zawartości *CAfile* i *CApath* i musi być jawnie wskazana przez wywołanie jednej z funkcji z rodziny *SSL_CTX_set_client_CA_list()* (tzn. także *SSL_set_client_CA_list()*, *SSL_CTX_add_client_CA()* lub *SSL_add_client_CA()*)

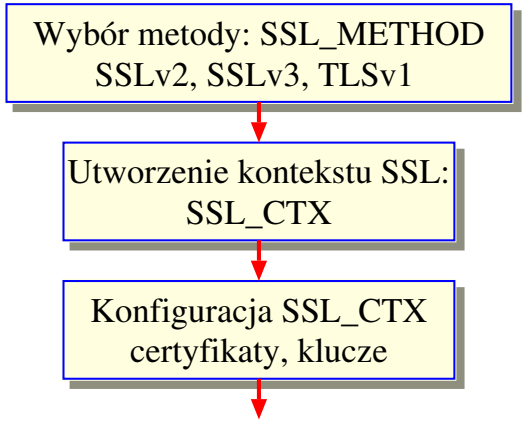
SSL API – wybór kluczy i certyfikatów (c.d.)

- Wczytanie certyfikatu CA – przykłady:

```
#include <openssl/ssl.h>
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile, const char *CApath);
```

- `ctx` wskazuje na strukturę `SSL_CTX`, do której zostanie wczytany certyfikat CA.
- Jeśli żądany certyfikat ma być określony przez plik (znajdujący się w tym samym katalogu co uruchamiana aplikacja), należy podać nazwę pliku w `CAfile`, a `CApath` ustawić na `NULL`.
- Jeśli certyfikat ma być określany przez katalog, `CAfile` powinno być ustawione na `NULL`, a katalog podany przez parametr `CApath`.

  Konieczne jest również utworzenie w podanym katalogu skrótów nazw przechowywanych tam certyfikatów („CA hash name") Skróty należy utworzyć programem `c_rehash`.



SSL API – wybór kluczy i certyfikatów – przykład

- Wczytanie certyfikatów i kluczy przez klienta:

```
#define CLIENT_CERT "client.pem"
#define CLIENT_KEY  "client.key"
#define CA_FILE     "ca.pem"

/*------  Wczytanie certyfikatu klienta do struktury SSL_CTX  ----------*/
if(SSL_CTX_use_certificate_file(ctx, CLIENT_CERT, SSL_FILETYPE_PEM) <= 0) {
    sys_err("Nie można wczytać certyfikatu klienta");
    exit(1);
}
/*------  Wczytanie klucza prywatnego do struktury SSL_CTX  ------------*/
if(SSL_CTX_use_PrivateKey_file(ctx, CLIENT_KEY, SSL_FILETYPE_PEM) <= 0) {
    sys_err("Nie można wczytać klucza prywatnego klienta");
    exit(1);
}
/* Wczytaj zaufane CA. */
if (!SSL_CTX_load_verify_locations(ctx, CA_CERT, NULL)) {
    sys_err("Błąd wczytywania listy CA");
    exit(1);
}
```

- Wczytany certyfikat CA (lub ich lista) zostanie użyty do weryfikacji certyfikatu serwera.
- Wczytany certyfikat klienta zostanie przedstawiony serwerowi. Klucz zostanie użyty jedynie do uwierzytelnienia certyfikatu.

SSL API – weryfikacja certyfikatów

- Certyfikat CA wczytany do struktury SSL_CTX jest używany do weryfikacji tożsamości drugiej strony. (Np. klient po połączeniu z serwerem weryfikuje certyfikat serwera – czy jest podpisany przez CA)
- Aby weryfikacja się udała, certyfikat drugiej strony musi być podpisany przez CA – bezpośrednio lub pośrednio (do weryfikacji pośredniej – łańcuch pośredników musi być kompletny)
- Maksymalna długość łańcucha od CA do weryfikowanego certyfikatu może być określona przez pole verify_depth struktury SSL_CTX lub SSL. Wartość w strukturze SSL jest dziedziczona z SSL_CTX podczas tworzenia struktury SSL za pomocą *SSL_new()*.
- Wartość 1 oznacza, że certyfikat musi być podpisany bezpośrednio przez CA. Wartość tę można ustawić wywołaniem funkcji *SSL_CTX_set_verify_depth*(ctx, głębokość):

```
#include <openssl/ssl.h>
void SSL_CTX_set_verify_depth(SSL_CTX *ctx,int depth);
void SSL_set_verify_depth(SSL *s, int depth);
```

- Sposób dokonywania weryfikacji certyfikatu można ustawić funkcją *SSL_CTX_set_verify()*:

```
#include <openssl/ssl.h>
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode, int (*verify_callback)(int, X509_STORE_CTX *));
void SSL_set_verify(SSL *s, int mode, int (*verify_callback)(int, X509_STORE_CTX *));
```

- Możliwe sposoby weryfikacji to:
  - SSL_VERIFY_NONE
  - SSL_VERIFY_PEER – najczęściej wykorzystywany sposób
  - SSL_VERIFY_FAIL_IF_NO_PEER_CERT
  - SSL_VERIFY_CLIENT_ONCE

SSL API – weryfikacja certyfikatów (c.d.)

parametr	tryb serwera	tryb klienta
SSL_VERIFY_NONE	serwer nie wysyła żądania podania certyfikatu do klienta, więc klient nie wyśle swojego certyfikatu	jeśli nie jest używane szyfrowanie *anonimowe* (bez uwierzytelniania – domyślnie zabronione), serwer wyśle certyfikat, który zostanie sprawdzony. Rezultat sprawdzenia można sprawdzić po wymianie (handshake) wołając *SSL_get_verify_result()*. Wymiana jest kontynuowana bez względu na wynik.
SSL_VERIFY_PEER	Serwer wysyła żądanie certyfikatu do klienta. Jeśli zostanie zwrócony, jest weryfikowany. Niepowodzenie weryfikacji przerywa *handshake* TLS/SSL i wysyła alert zawierający przyczynę niepowodzenia. Można to zmienić modyfikując znaczniki SSL_VERIFY_FAIL_IF_NO_PEER_CERT i SSL_VERIFY_CLIENT_ONCE.	Certyfikat serwera jest weryfikowany. Niepowodzenie weryfikacji przerywa *handshake* TLS/SSL generując alert. Jeśli nie otrzymano certyfikatu serwera, bo wykorzystywane jest szyfrowanie anonimowe, parametr SSL_VERIFY_PEER jest ignorowany.
SSL_VERIFY_FAIL_IF_NO_PEER_CERT	Brak certyfikatu od klienta przerywa *handshake* TLS/SSL i generuje alert handshake failure. Parametr musi być używany wraz z SSL_VERIFY_PEER.	ignorowany
SSL_VERIFY_CLIENT_ONCE	Żądaj certyfikatu klienta tylko przy pierwszej negocjacji (przy renegocjacjach już nie). Parametr musi być używany wraz z SSL_VERIFY_PEER.	ignorowany

SSL API – weryfikacja certyfikatów (c.d.)

Alternatywna metoda – *SSL_get_verify_result()*

```
# include <openssl/ssl.h>
long SSL_get_verify_result(const SSL *ssl);
```

- Funkcja zwraca wynik weryfikacji certyfikatu X509 drugiej strony, jeśli został przedstawiony.
- Jeśli wystąpi kilka błędów weryfikacji, raportowany jest tylko ostatni.
- Jeśli druga strona nie przedstawiła żadnego certyfikatu, to zwracany jest kod `X509_V_OK`, ponieważ nie nastąpiła żadna weryfikacja, nie było więc żadnego błędu.
- Ze względu na ostatni punkt – funkcja jest użyteczna tylko wraz z *SSL_get_peer_certificate()*

Sposób wykorzystania *SSL_get_verify_result()*:

1. Wywołanie *SSL_connect()* (klient) lub *SSL_accept()* (serwer), aby nawiązać połączenie SSL (SSL handshake) i dokonać weryfikacji certyfikatów.
2. Wywołanie *SSL_get_peer_certificate()*, aby w sposób jawny uzyskać certyfikat drugiej strony

SSL API – *SSL_get_verify_result()* – przykład wykorzystania

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

SSL_CTX_set_verify_depth(ctx, 1);
err = SSL_connect(ssl);

if (SSL_get_peer_certificate(ssl) != NULL) {
    if (SSL_get_verify_result(ssl) == X509_V_OK)
        BIO_printf(bio_c_out, "Weryfikacja klienta za pomocą SSL_get_verify_result() udana.\n");
    else {
        BIO_printf(bio_err, "Weryfikacja klienta za pomocą SSL_get_verify_result() NIE POWIODŁA SIĘ.\n");
        exit(1);
    }
} else
    BIO_printf(bio_c_out, "druga strona nie przedstawiła certyfikatu!\n");
```

SSL API – wybór szyfrów (opcjonalny)

- Wybór szyfrów nie jest konieczny – wystarcza wybór metody (wersji SSL/TLS)

```
int SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str);
int SSL_set_cipher_list(SSL *ssl, const char *str);

STACK_OF(SSL_CIPHER) *SSL_get_ciphers(const SSL *ssl);
const char *SSL_get_cipher_list(const SSL *ssl, int priority);

const char *SSL_CIPHER_get_name(const SSL_CIPHER *cipher);
```

- Lista szyfrów w łańcuchu tekstowym w formacie zwracanym przez komendę `openssl ciphers`

```
% openssl ciphers
DHE-RSA-AES256-SHA:DHE-DSS-AES256-SHA:AES256-SHA:EDH-RSA-DES-CBC3-SHA:EDH-DSS-DES-CBC3-SHA:DES-CBC3-SHA:
DES-CBC3-MD5:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA:AES128-SHA:RC2-CBC-MD5:RC4-SHA:RC4-MD5:RC4-MD5:
EDH-RSA-DES-CBC-SHA:EDH-DSS-DES-CBC-SHA:DES-CBC-SHA:DES-CBC-MD5:EXP-EDH-RSA-DES-CBC-SHA:
EXP-EDH-DSS-DES-CBC-SHA:EXP-DES-CBC-SHA:EXP-RC2-CBC-MD5:EXP-RC2-CBC-MD5:EXP-RC4-MD5:EXP-RC4-MD5
```

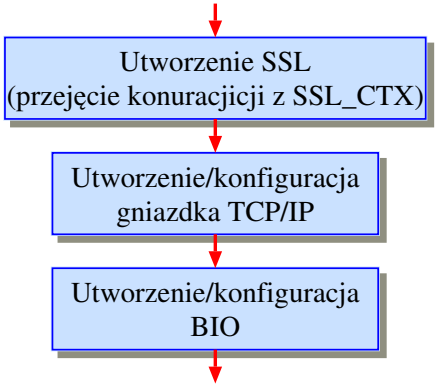
- Niektóre z szyfrów mogą wymagać ustawienia odpowiedniej funkcji callbackowej do ustawienia tymczasowego klucza sesji, np. *SSL_CTX_set_tmp_rsa_callback()* lub *SSL_CTX_set_tmp_dh_callback()*
- Wybór konkretnego szyfru następuje w chwili negocjacji połączenia SSL (SSL handshake). Wybrany szyfr musi być dostępny po obu stronach połączenia
- Zbytnie ograniczanie listy szyfrów może skutkować niemożnością połączenia (brak wspólnych elementów na listach)

SSL API – utworzenie SSL z kontekstu SSL_CTX

```
#include <openssl/ssl.h>
SSL* SSL_new(SSL_CTX *ctx);
```

- Nowo utworzona struktura SSL dziedziczy wszystkie ustawienia ze struktury SSL_CTX – metody łączenia, ustawienia weryfikacji, certyfikaty, timeouty, itp.
- Jeśli wymagane ustawienia zostały dokonane w strukturze SSL_CTX, to w SSL nie trzeba już robić niczego więcej
- Można jednak zmieniać odziedziczone ustawienia za pomocą odpowiednich funkcji API.

  Przykład: Funkcja *SSL_CTX_use_certificate()* wczytuje certyfikat do struktury SSL_CTX – analogicznie: *SSL_use_certificate()* wczytuje go do struktury SSL.



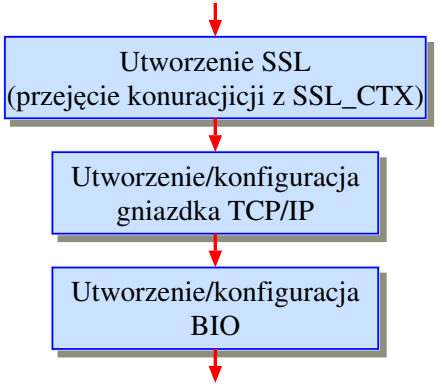
Nawiązanie połączenia TCP

- Nawiązanie połączenia TCP odbywa się „standardowymi” metodami
- Serwer nawiązuje połączenie funkcją *accept()*:

```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);

if (bind(sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
    perror("serwer: nie można nadać gniazdku adresu");
listen(sock, 5);
clen = sizeof(cl_addr);
newsock = accept(sock, (struct sockaddr*) &cl_addr, &clen);
if (newsock < 0)
    perror("serwer: błąd w funkcji accept()");
```

- klient wywołuje *connect()*:

```
if ((sock=socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror("klient: nie można utworzyć gniazdka");
if (connect(sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
    perror("klient: nie można połączyć się z serwerem);
```



Powiązanie gniazdka TCP z SSL

- Połączone gniazdko powinno zostać przekazane do struktury SSL lub BIO
- Najprościej – wiążąc gniazdko ze strukturą SSL:

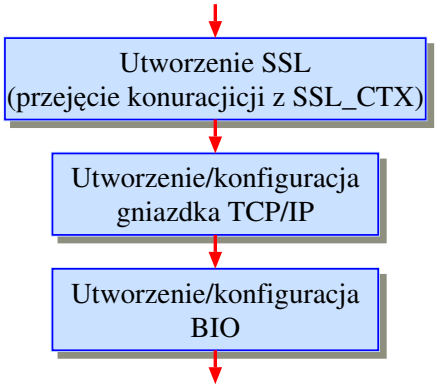
```
SSL_set_fd(ssl, socket);  // socket -- połączone gniazdko TCP (connect/accept)
```

- Polecany sposób: przez strukturę BIO (buffered i/o):

```
sbio=BIO_new(BIO_s_socket());
BIO_set_fd(sbio, socket, BIO_NOCLOSE); // socket -- połączone gniazdko TCP
SSL_set_bio(ssl, sbio, sbio);
```

alternatywna, równoważna metoda:

```
sbio = BIO_new_socket(socket, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
```



SSL API – SSL handshake

„SSL handshake" to negocjacja parametrów połączenia pomiędzy klientem a serwerem SSL połączona z wymianą kluczy.

- Po stronie serwera:

```
error = SSL_accept(ssl);
```

- Po stronie klienta:

```
error = SSL_connect(ssl);
```

Alternatywnie: za pomocą *SSL_read()* i *SSL_write()*:

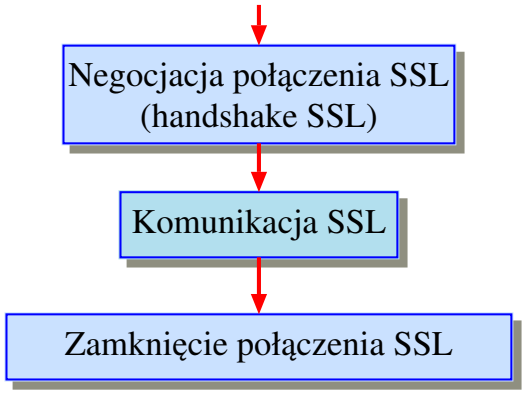
- Po stronie serwera:

```
error = SSL_accept_state(ssl);
SSL_read(ssl, buf, size);
```

- Po stronie klienta:

```
error = SSL_set_connect_state(ssl);
SSL_write(ssl, buf, size);
```

- Opcjonalnie: wywołanie *SSL_get_peer_certificate*(ssl) w celu uzyskania certyfikatu drugiej strony (np. aby sprawdzić datę ważności, nazwę, itp.)



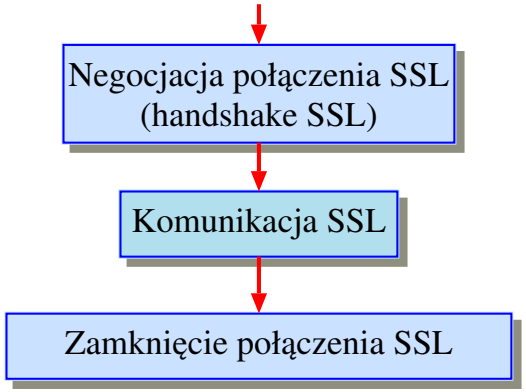
SSL API – SSL komunikacja

Po dokonaniu negocjacji (handshake) można korzystać z bezpiecznego połączenia.

- *SSL_read()* służy do odczytu danych, *SSL_write()* do zapisu – w analogiczny sposób jak byłyby wykorzystane „zwykłe" funkcje *read()* i *write()*
- Można także używać *BIO_puts()* i *BIO_gets()* oraz *BIO_write()* i *BIO_read()*, jeśli wcześniej zostanie utworzony bufor BIO:

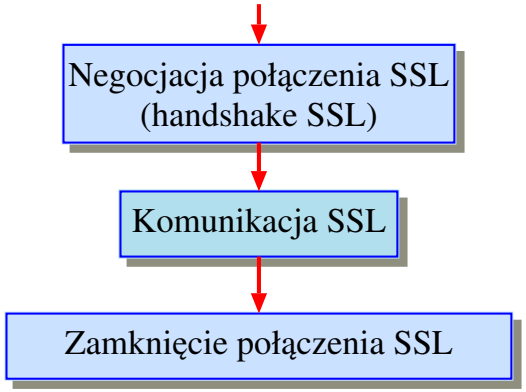
```
BIO *buf_io, *ssl_bio;
char rbuf[R_SIZE];
char wbuf[W_SIZE];

buf_io=BIO_new(BIO_f_buffer());       // utwórz bufor BIO
ssl_bio=BIO_new(BIO_f_ssl());         // utwórz ssl BIO

BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE); // przypisz ssl BIO do SSL
BIO_push(buf_io, ssl_bio);            // dodaj ssl_bio do buf_io

ret=BIO_puts(buf_io, wbuf);           // zapisz zawartość wbuf do buf_io
ret=BIO_gets(buf_io, rbuf, R_SIZE);   // przeczytaj dane z buf_io do rbuf
ret=BIO_write(buf_io, wbuf, n);       // zapisz n bajtów z wbuf do buf_io
ret=BIO_read(buf_io, rbuf, rlen);     // przeczytaj rlen danych z buf_io do rbuf

BIO_free(buf_io);
```

- W dowolnym momencie jedna ze stron może zażądać renegocjacji połączenia lub jego zamknięcia



SSL API – zamykanie połączenia SSL

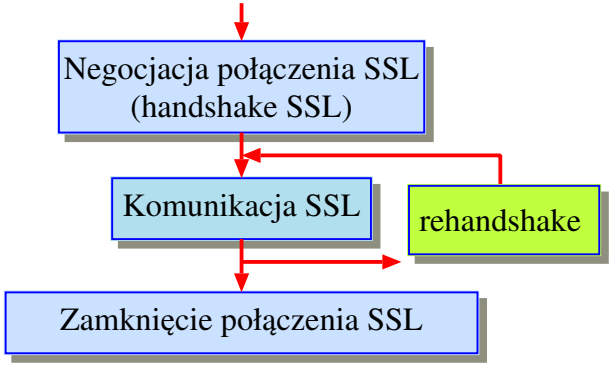
- Zamknięcie następuje z dowolnej strony (klienta lub serwera) przez wywołanie *SSL_shutdown()*
- Pierwsze wywołanie funkcji wysyła alert 'close notify' i ustawia flagę `SSL_SENT_SHUTDOWN`, funkcja zwraca wartość 0, a połączenie zostaje jednokierunkowo zamknięte.
- Drugie wywołanie *SSL_shutdown*(ssl) (opcjonalne) czeka na odebranie 'close notify' od drugiej strony, po czym zwraca wartość 1.
- Jeśli druga strona już wcześniej zamknęła połączenie (co zostało obsłużone przy okazji *SSL_read()*), ustawiona jest flaga `SSL_RECEIVED_SHUTDOWN`, a *SSL_shutdown()* od razu zwraca 1. Stan tej flagi można sprawdzić wołając *SSL_get_shutdown()*.
- dwustronnie zamknięte połączenie może być ponownie wykorzystane do nowej komunikacji SSL
- jednostronne zamknięcia połączenia SSL wystarcza, jeśli gniazdko TCP służące do kumunikacji i tak ma być zaraz zamknięte. Jeśli planowane jest ponowne wykorzystanie SSL – konieczne jest zamknięcie dwustronne.



SSL API – renegocjacja połączenia SSLv3

- Renegocjację połączenia w SSLv3/TLSv3 może rozpocząć zarówno klient jak i serwer.
- SSLv2 nie dopuszcza renegocjacji.
- Serwer woła *SSL_renegotiate()* a następnie dwukrotnie *SSL_do_handshake()*.
  - *SSL_renegotiate()* jedynie ustawia odpowiednie znaczniki w połączeniu.
  - Pierwsze wywołanie *SSL_do_handshake()* wysyła „Server hello" do klienta – jeśli zakończy się sukcesem, oznacza to, że klient zgodził się na renegocjację
  - serwer ustawia wówczas w strukturze SSL stan `SSL_ST_ACCEPT` i ponownie woła *SSL_do_handshake()*, aby dokończyć renegocjację.
- Klient woła *SSL_renegotiate()* a następnie *SSL_do_handshake()* (tylko raz)
  - *SSL_renegotiate()* ustawia odpowiednie znaczniki
  - *SSL_do_handshake()* przeprowadza cały proces renegocjacji.
- Przykład kodu serwera:

```
if(SSL_renegotiate(ssl) <= 0) {
  printf("SSL_renegotiate() failed\n");    exit(1);
}
if(SSL_do_handshake(ssl) <= 0) {
  printf("SSL_do_handshake() failed\n");   exit(1);
}
ssl->state = SSL_ST_ACCEPT;
if(SSL_do_handshake(ssl) <= 0) {
  printf("SSL_do_handshake() failed\n");   exit(1);
}
```

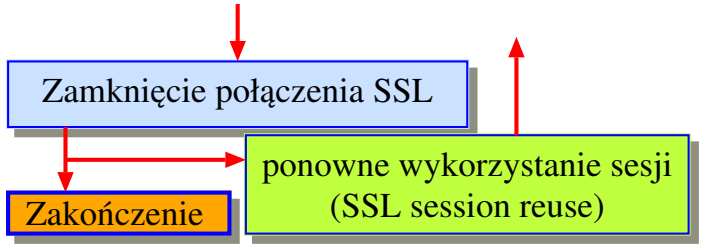


SSL API – kończenie wykorzystywania SSL

- Po zamknięciu połączenia przez *SSL_shutdown*() pozostaje posprzątać
- Sprzątanie polega na zwolnieniu pamięci używanej przez różnego rodzaju struktury danych w porocesie łączenia/negocjacji itd.
- Jeśli była wołana jakaś funkcja typu *xxx_new*() (np. *SSL_new*(), *BIO_new*(), itp.), powinna zostać wywołana odpowiednia funkcja *xxx_free*().
- Struktury tworzone niejawnie są automatycznie usuwane, gdy obiekt z nimi przestaje istnieć.

  Np. struktury BIO utworzone podczas wywołania *SSL_new*() zostaną automatycznie usunięte przy wywołaniu *SSL_free*() – nie trzeba ich zwalniać za pomocą *BIO_free*().

  UWAGA! – *SSL_free*() może zostać wywołane dopiero po wcześniejszym wywołaniu *SSL_shutdown*()
- Gniazdka TCP połączone wcześniej za pomocą *connect*()/*accept*() wykorzystywane do komunikacji SSL należy rozłączyć w standardowy sposób (*shutdown*().



Program `tlswrap`

- Aplikacja pozwalająca wykorzystać dowolnego klienta FTP do łączenia z serwerem FTP obsługującym SSL.
- Standardowo – nasłuchuje lokalnie na porcie 7000 i działa jak proxy FTP
- Klient FTP po połączeniu się z proxy podaje nazwę użytkownika w postaci **username@hostname:port**, a następnie hasło.
- `tlswrap` jako proxy nawiąże bezpiecznie połączenie z użyciem SSL do swerwera hostname na port port, próbując tam autoryzować użytkownika **username**.

Przy okazji – dobre źródło przykładów programowania SSL/TLS

- Łącznie ok. 3200 linii kodu, ale to:
  - 500 linii kodu SSL/TLS
  - 400 linii komunikacji sieciowej na poziomie warstwy TCP/IP
  - 1500 linii obsługi parametrów linii komend, konfiguracji, samego serwera proxy
  - 700 linii różnych funkcji pomocniczych