



Obrona

# Szyfrowanie w aplikacjach – biblioteka Openssl

Paweł Maziarz 

stopień trudności



**Informacja w dzisiejszych czasach to jedna z bardziej cennych rzeczy, dlatego trzeba o nią odpowiednio zadbać. Sieci firmowe, osiedlowe czy bezprzewodowe często nie są zabezpieczone na odpowiednim poziomie (nieraz nie pozwalają na to przyczyny techniczne), a więc przesyłając informację w sieci, jej nienaruszalność powinna zagwarantować aplikacja.**

**N**ajbardziej popularnym i skutecznym rozwiązaniem tego, zdaje się cały czas okazywać szyfrowanie SSL (ang. Secure Socket Layer) lub rozwijane dalej pod nazwą TLS (ang. Transport Layer Security) za pomocą wolnodostępnej biblioteki OpenSSL, co wykorzystuje większość kluczowych aplikacji sieciowych jak Apache, Postfix, OpenSSH. Pakiet OpenSSL składa się generalnie z trzech składników:

- biblioteka libssl – obsługuje protokół SSL/TLS;
- biblioteka libcrypto – zawiera obsługę algorytmów kryptograficznych;
- polecenie openssl – narzędzie administracyjne pozwalające w linii poleceń skorzystać z wymienionych wyżej bibliotek.

Protokół SSL/TLS składa się z trzech faz:

- negocjacja algorytmów;
- wymiana kluczy symetrycznych przy pomocy infrastruktury klucza publicznego (PKI) i uwierzytelnianiu opartym na certyfikatach;
- szyfrowanie symetryczne za pomocą wcześniej wymienionych kluczy.

W pierwszej fazie klient i serwer wymieniają się informacjami odnośnie wersji biblioteki oraz wspieranych algorytmów. W drugiej fazie serwer uwierzytelnia się klientowi przedstawiając swój certyfikat podpisany przez klucz prywatny. Klient mając już klucz publiczny serwera może stwierdzić czy ma do czynienia z właściwym serwerem. Opcjonalnie może jeszcze wystąpić analogiczne uwierzytelnianie klienta. Infrastruktura klucza publicznego – PKI (ang. Public Key Infrastructure) to szyfrowanie asymetryczne, które jest dużo mniej wydajne niż szyfrowanie symetryczne, dlatego też następnym krokiem tej fazy jest wymiana kluczy symetrycznych (wygenerowanych na przykład za pomocą al-

## Z artykułu dowiesz się...

- jak zaimplementować szyfrowanie SSL/TLS w swoich aplikacjach

## Co powinieneś wiedzieć...

- powinieneś umieć programować w języku C

gorytmu Diffiego-Hellmana), które będą używane we właściwym przesyłaniu danych. Ostatnia faza to już faktyczne przesyłanie danych zaszyfrowanych wcześniej ustalonym kluczem symetrycznym.

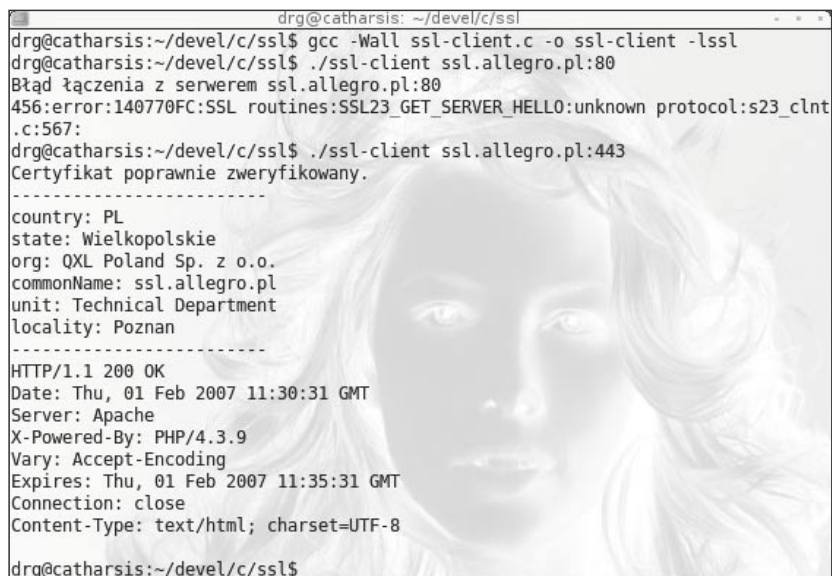
## Certyfikaty

Jak można wywnioskować z poprzedniego akapitu, zanim stworzymy klienta i serwer z obsługą szyfrowania SSL/TLS, trzeba najpierw przygotować kilka certyfikatów, z których nasze programy będą korzystać. W naszym przypadku będą potrzebne dwa – certyfikat Urzędu Certyfikującego (ang. Certification Authority – CA), którym będziemy podpisywać certyfikaty klientów oraz certyfikat dla samego klienta. Do ich stworzenia posłuży komenda `openssl`. By ułatwić cały proces, stworzymy plik `ssl.cnf` (Listing 1), którego omawiać nie będę, ponieważ dostarczony w całym pakiecie plik konfiguracyjny `openssl.cnf` zaopatrzony jest w szczegółowe komentarze. Musimy też stworzyć odpowiednią strukturę katalogów i pewne pliki:

```
drg@catharsis:~$ mkdir -p ssl/
{certs,private}
drg@catharsis:~$ chmod 700 ssl/private/
drg@catharsis:~$ touch ssl/index.txt
drg@catharsis:~$ echo 01 > ssl/serial
```

### Listing 1. `Ssl.cnf` - przykładowy plik konfiguracyjny dla `openssl`

```
[ ca ]
default_ca = CA_kajmany
[ CA_kajmany ]
dir = .
certs = $dir/certs
new_certs_dir = $dir/certs
private_key = $dir/private/akey.pem
serial = $dir/serial
database = $dir/index.txt
certificate = $dir/cacert.pem
default_days = 364
default_md = sha1
policy = kajmany_policy
x509_extensions = kajmany_ext
[ kajmany_policy ]
commonName = supplied
countryName = supplied
stateOrProvinceName = supplied
organizationName = supplied
organizationalUnitName = supplied
emailAddress = optional
[ kajmany_ext ]
basicConstraints = CA:false
[ req ]
distinguished_name = kajmany_dn
default_bits = 1024
default_keyfile = ./private/akey.pem
x509_extensions = kajmany_ca_ext
prompt = no
[ kajmany_dn ]
commonName = intersim.pl
countryName = PL
stateOrProvinceName = Dolnoslaskie
localityName = Wroclaw
organizationName = Intersim s.c.
organizationalUnitName = Centrum Certyfikatow
emailAddress = pawel.maziarz@intersim.pl
[ kajmany_ca_ext ]
basicConstraints = CA:false
```



```
drg@catharsis:~/devel/c/ssl$ gcc -Wall ssl-client.c -o ssl-client -lssl
drg@catharsis:~/devel/c/ssl$ ./ssl-client ssl.allegro.pl:80
Błąd łączenia z serwerem ssl.allegro.pl:80
456:error:140770FC:SSL routines:SSL23_GET_SERVER_HELLO:unknown protocol:s23_clnt
.c:567:
drg@catharsis:~/devel/c/ssl$ ./ssl-client ssl.allegro.pl:443
Certyfikat poprawnie zweryfikowany.
-----
country: PL
state: Wielkopolskie
org: QXL Poland Sp. z o.o.
commonName: ssl.allegro.pl
unit: Technical Department
locality: Poznan
-----
HTTP/1.1 200 OK
Date: Thu, 01 Feb 2007 11:30:31 GMT
Server: Apache
X-Powered-By: PHP/4.3.9
Vary: Accept-Encoding
Expires: Thu, 01 Feb 2007 11:35:31 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
drg@catharsis:~/devel/c/ssl$
```

Rysunek 1. Kompilacja i uruchomienie klienta

po czym można przystąpić do generowania certyfikatu głównego (CA) komendą;

```
drg@catharsis:~/ssl$ openssl req -x509
-newkey rsa:2048 -out cacert.pem
-outform PEM -config ssl.cnf
```

Po podaniu i potwierdzeniu hasła do certyfikatu, zostanie stworzony certyfikat Urzędu Certyfikującego `ca.cert.pem` oraz jego klucz prywatny w `private/akey.pem`. Teraz trzeba stworzyć certyfikat dla klienta, który będzie używany w późniejszych aplikacjach. Certyfikat ten następnie będzie musiał być podpisany przez nasze CA. Żądanie certyfikatu klient może utworzyć u siebie po czym wysłać go do Urzędu Certyfikującego



w celu podpisania, który po podpisaniu odeśle mu już pełnowartościowy certyfikat, Urząd Certyfikujący może też mieć politykę, że sam generuje certyfikaty i od razu podpisuje je dla klientów, co będzie miało miejsce w naszym przykładzie. Żądanie certyfikatu:

```
drg@catharsis:~/ssl$ openssl req -
newkey rsa:2048 -keyout clients/cert-
server-key.pem -keyform PEM -out
clients/cert-server-request.pem -
outform PEM
```

i jego podpisanie:

```
drg@catharsis:~/ssl$ openssl ca -
in clients/cert-server-request.pem
-notext -out clients/cert-server.pem
-config ssl.cnf
```

Wypada jeszcze wygenerować dla serwera plik z parametrami dla algorytmu Diffiego-Hellmana poleceniem:

```
drg@catharsis:~/ssl$ openssl dhparam
-out clients/dhserver.pem 512
```

Po tych czynnościach w katalogu *clients/* znajduje się gotowy certyfikat do wykorzystania w aplikacji *cert-server.pem*, klucz prywatny do niego *cert-key.pem* oraz plik *dhserver.pem* z parametrami DH. Plik *cert-server-request.pem* nie jest już potrzebny, należy go zatem usunąć. Informacje o stworzonych certyfikatach możemy wyświetlić komendami:

### Listing 2. Funkcja *ssl\_client\_initialize\_ctx* z pliku *ssl-client.c*

```
SSL_CTX *ssl_client_initialize_ctx() {
    SSL_CTX *ctx;
    SSL_METHOD *method;
    SSL_library_init();
    OpenSSL_add_all_algorithms();
    SSL_load_error_strings();
    ERR_load_crypto_strings();
    method = SSLv23_client_method();
    ctx = SSL_CTX_new(method);
    if (!SSL_CTX_load_verify_locations(ctx, CA_FILE, NULL) || !
        SSL_CTX_set_default_verify_paths(ctx)) {
        fprintf(stderr, "Błąd w ładowaniu certyfikatu CA, nie zweryfikuję zatem
            certyfikatu serwera.\n");
        ERR_print_errors_fp(stderr);
    }
    SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);
    SSL_CTX_set_verify_depth(ctx, 1);
    SSL_CTX_set_options(ctx, SSL_OP_ALL | SSL_OP_NO_SSLv2);
    return ctx;
}
```

### Listing 3. *bioex.c* – przykład łańcucha BIO

```
#include <openssl/bio.h>
#include <openssl/ssl.h>
int main() {
    BIO *bio, *b64;
    char message[] = "Bio przykład";
    bio = BIO_new(BIO_s_file());
    BIO_set_fp(bio, stdout, BIO_NOCLOSE);
    b64 = BIO_new(BIO_f_base64());
    bio = BIO_push(b64, bio);
    BIO_write(bio, message, strlen(message));
    BIO_flush(bio);
    BIO_free_all(bio);
    return 0;
}
```

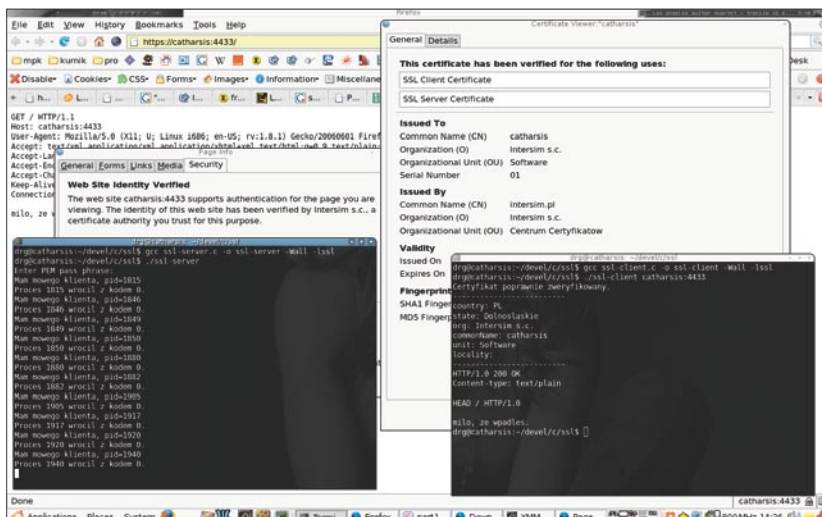
```
drg@catharsis:~/ssl$ openssl req -
noot -text -in clients/cert-server-
request.pem
drg@catharsis:~/ssl$ openssl x509 -
```

```
noot -text -in clients/cert-server.pem
drg@catharsis:~/ssl$ openssl x509 -
noot -text -in cacert.pem
```

## Budujemy klienta ssl

Na pierwszy ogień pójdzie aplikacja klienta, która zestawia szyfrowane połączenie z serwerem, wysyła zapytanie HTTP (by od razu można było przetestować na realnych serwerach WWW z obsługą połączeń SSL/TLS), oraz wyświetla informacje o certyfikacie drugiej strony i odczytuje odpowiedź.

Na samym początku, aplikacja *openssl* powinna zainicjować dostępne algorytmy, szyfry oraz nazwy błędów, które w razie problemów będą bardziej pomocne niż ich liczbowe wartości, wywołując funkcje:



Rysunek 2. Kompilacja i działanie serwera

```

SSL_library_init();
OpenSSL_add_all_algorithms();
SSL_load_error_strings();
ERR_load_crypto_strings();

```

Następnie trzeba określić jakimi protokołami aplikacja będzie się porozumiewać oraz jaką rolę w tej komunikacji będzie spełniać – serwera, klienta czy też może obie. Określa się to poprzez zainicjowanie struktury typu `SSL_METHOD` \*method odpowiednią funkcją, której pierwszym członem jest typ protokołu, jeden z:

- SSLv2 – SSL w wersji 2, niezalecane z powodu poważnych luk bezpieczeństwa;
- SSLv3 – SSL w wersji 3;
- TLSv1 – TLS w wersji 1.0;
- SSLv23 – SSL w wersji 2, 3 lub TLS w wersji 1.0
- DTLSv1 – DTLS w wersji 1.0 (szyfrowanie datagramów pakietów, np. UDP).

Jeżeli chodzi więc o kompatybilność widać, że najlepszym wyborem będzie SSLv23, jednak druga strona łatwo może wymusić protokół SSL 2, co narazi na niebezpieczeństwo nasz programowany system. Można sobie jednak z tym poradzić w późniejszym czasie, poprzez przekazanie jednej z opcji `SSL_OP_NO_SSLv2`, `SSL_OP_NO_SSLv3` albo `SSL_OP_NO_TLSv1` do konkretnego połączenia lub całego kontekstu SSL, która wyłączy obsługę konkretnego protokołu. Następnym członem jest rola – *client*, *server* lub brak, jeżeli aplikacja ma pracować zarówno jako klient i serwer. Dla klienta TLS inicjalizacja struktury *method* będzie więc miała postać `SSL_METHOD *method = TLSv1_client_method()`, dla serwera `SSL_METHOD *method = TLSv1_server_method()`, a dla aplikacji hybrydowej `SSL_METHOD *method = TLSv1_method()`.

Po wyborze metody czas na zainicjowanie całego kontekstu SSL (struktura `SSL_CTX` \*ctx). Kontekst ten musi być całkowicie przygotowany przed otwarciem

pierwszego połączenia. Każde nowe połączenie w aplikacji będzie odnosiło się do tego jednego kon-

tekstu, należy więc pamiętać, że zmieniając właściwości kontekstu, np. `SSL_CTX_set_options(ctx,`

**Listing 4.** główna funkcja z pliku *ssl-client.c*

```

#define HOSTPORT    "localhost:4433"
#define CA_FILE     "/home/drg/ssl/cacert.pem"
#define REQUEST    "HEAD / HTTP/1.0\r\n\r\n"
void certificate_print_info(SSL *ssl);
SSL_CTX *ssl_client_initialize_ctx();
int main(int argc, char **argv) {
    char *host_port;
    BIO *sbio;
    SSL_CTX *ctx;
    SSL *ssl;
    char buf[1024];
    int len;
    long int err;
    if (argc > 1)
        host_port = strdup(argv[1]);
    else
        host_port = strdup(HOSTPORT);
    ctx = ssl_client_initialize_ctx();
    sbio = BIO_new_ssl_connect(ctx);
    BIO_set_conn_hostname(sbio, host_port);
    BIO_get_ssl(sbio, &ssl);
    if (!ssl) {
        fprintf(stderr, "Błąd ssl.\n");
        ERR_print_errors_fp(stderr);
        return 1;
    }
    SSL_set_mode
    (ssl, SSL_MODE_AUTO_RETRY);
    if (BIO_do_connect(sbio) <= 0) {
        fprintf(stderr, "Błąd łączenia
z serwerem %s\n", host_port);
        ERR_print_errors_fp(stderr);
        return 1;
    }
    if ((err = SSL_get_verify_result(ssl)) !=
X509_V_OK) {
        fprintf(stderr, "Błąd przy weryfikacji
certyfikatu (kod %ld - %s).\n",
err, X509_verify_cert_error_string(err));
        ERR_print_errors_fp(stderr);
    }
    else
        fprintf(stderr,
"Certyfikat poprawnie zweryfikowany.\n");
    certificate_print_info(ssl);
    if (BIO_do_handshake(sbio) <= 0) {
        fprintf(stderr,
"Error establishing SSL connection\n");
        ERR_print_errors_fp(stderr);
        return 1;
    }
    BIO_puts(sbio, REQUEST);
    while(1) {
        len = BIO_read(sbio, buf, 1024);
        if (len <= 0) break;
        write(STDOUT_FILENO, buf, len);
    }
    BIO_free_all(sbio);
    exit(0);
}

```



**Listing 5. Funkcja `certificate_print_info` z `ssl_client.c`**

```

void certificate_print_info(SSL *ssl) {
    X509 *cert;
    char commonName[128];
    char country[128];
    char state[128];
    char locality[128];
    char org[128];
    char unit[128];
    *commonName = *country = *state = *locality = *org = *unit = '\0';
    cert = SSL_get_peer_certificate(ssl);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_countryName,
                              country, 128);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_
                              stateOrProvinceName, state,
    128);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_localityName,
                              locality,
    128);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_organizationName,
                              org,
    128);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_commonName,
                              commonName, 128);
    X509_NAME_get_text_by_NID(X509_get_subject_name(cert), NID_
                              organizationalUnitName,
    unit, 128);
    X509_free(cert);
    fprintf(stderr, "\n");
    fprintf(stderr, "country: %s\n", country);
    fprintf(stderr, "state: %s\n", state);
    fprintf(stderr, "org: %s\n", org);
    fprintf(stderr, "commonName: %s\n", commonName);
    fprintf(stderr, "unit: %s\n", unit);
    fprintf(stderr, "locality: %s\n", locality);
    fprintf(stderr, "\n");
}

```

SSL\_OP\_ALL | SSL\_OP\_NO\_SSLv2) zmiana będzie dotyczyć wszystkich połączeń, podczas gdy ustawienie tych opcji tylko dla jednego konkretnego połączenia miałyby postać `SSL_set_options(sslcon, SSL_OP_ALL | SSL_OP_NO_SSLv2)`. Za inicjalizację kontekstu odpowiedzialne jest wywołanie `SSL_CTX *ctx = SSL_CTX_new(method)`, gdzie `method` zostało omówione wcześniej. W bardziej zwartej wersji można to oczywiście zapisać `ctx = SSL_CTX_new(SSLv23_client_method())` (w przypadku klienta). Wszystko już dla klienta zostało prawie przygotowane, w celu weryfikacji jednak certyfikatów serwerów, z którymi będzie się klient łączył, należy jeszcze załadować certyfikat Urzędu Certyfikują-

cego (CA), zrobimy to używając funkcji `SSL_CTX_load_verify_locations()` oraz `SSL_CTX_set_default_verify_paths()`. Przyda się też określić parametry weryfikacji, służy do tego funkcja `SSL_CTX_set_verify()`. Pierwszym parametrem jest oczywiście wskaźnik na kontekst (widać to po przedrostku `SSL_CTX_`), następnym jest flaga określająca zasad weryfikacji, dostępne flagi dla klienta to:

- `SSL_VERIFY_NONE` – przesłany certyfikat od serwera będzie sprawdzony, jednak niezależnie od wyniku, połączenie będzie kontynuowane; wynik weryfikacji można później sprawdzić za pomocą funkcji `SSL_get_verify_result()`;

- `SSL_VERIFY_PEER` – jeżeli weryfikacja certyfikatu serwera nie powiedzie się, połączenie jest natychmiastowo zatrzymywane z informacją o błędzie.

Trzecim i ostatnim argumentem tej funkcji jest wskaźnik na własną funkcję weryfikującą, z której można jednak zrezygnować podając jako argument `NULL`, co poczynimy w naszym kliencie. Przy ustawianiu parametrów weryfikacji, możemy ustawić jeszcze limit głębokości sprawdzanych certyfikatów (domyślnie jest 9) za pomocą funkcji `SSL_CTX_set_verify_depth()`. Ustawienie głębokości na 1 spowoduje, że sprawdzony będzie tylko certyfikat serwera (Urzędu Certyfikującego już nie), co w naszym przypadku zupełnie wystarczy.

Opisane czynności inicjujące umieściliśmy w naszym kliencie w osobnej funkcji `ssl_client_initialize_ctx()` przedstawionej na Listingu 2.

### Basic Input/Output

BIO to abstrakcja wejścia/wyjścia dostępna dla programisty `openssl`. Dzięki BIO aplikacja może schować wszystkie niskopoziomowe wywołania związane z obsługą połączenia SSL, z obsługą nieszyfrowanego połączenia TCP/IP czy też z obsługą plików. Istnieją dwa rodzaje BIO

- `source/sink BIO`
- `filter BIO`

### O autorze

Autor jest właścicielem i jednocześnie jednym z głównych programistów firmy tworzącej między innymi sieciowe oprogramowanie. Na przełomie ostatnich lat współpracował z kilkoma firmami w charakterze Security Specialist. W wolnych chwilach gra w golfa, na gitarze klasycznej oraz spuszcza się na linie z budynków. Kontakt z autorem: [pawel.maziarz@intersim.pl](mailto:pawel.maziarz@intersim.pl).

Pierwsze z nich służą do bezpośredniego operowania na gniazdach czy plikach. Drugie – filtrujące – pobierają dane z innych BIO i przekazują je do następnych lub wprost do aplikacji. Dane te mogą zostać niezmienione (np. BIO buforujące) bądź odpowiednio przetworzone (np. BIO szyfrujące). Ciekawą rzeczą jest fakt, że BIO mogą być łączone w łańcuchy (pojedyncze BIO to łańcuch z jednym elementem) za pomocą funkcji `BIO_push()` (intuicyjnie `BIO_pop()` zdejmuje BIO z łańcucha), dzięki czemu w łatwy i praktycznie przezroczysty sposób można dokonywać operacji na danych. Do tworzenia abstrakcyjnych BIO służy funkcja `BIO_new()` przyjmująca jako parametr rodzaj tworzonego BIO (dostępne rodzaje można znaleźć na stronie podręcznika *man bio*). Szybka demonstracja łańcuchów BIO przedstawiona jest na Listingu 3.

Program ten tworzy nowe BIO plikowe (jest to BIO typu `source/sink`), po czym przypisuje do niego standardowe wyjście:

```
bio = BIO_new(BIO_s_file());
BIO_set_fp(bio, stdout, BIO_
            NOCLOSE);
```

i tutaj mała uwaga – studiując stronę podręcznika *man BIO\_s\_file* można zauważyć, że zamiast tych dwóch linijek można użyć funkcji `BIO_new_fp()`, która zrobi dokładnie to samo, a więc prościej:

```
bio = BIO_new_fp(stdout, BIO_NOCLOSE);
```

W następnych linijkach program tworzy nowe BIO, tym razem filtrujące, typu `BIO_f_base64()`. Jest to BIO, które koduje wszystko co jest do niego zapisane algorytmem `base64`, oraz dekoduje w przypadku odczytu z niego. Funkcja `BIO_push(b64, bio)` dołącza do łańcucha złożonego do tej pory z BIO plikowego typu `source/sink` BIO filtrujące `b64` i zwraca łańcuch `bio`. Od tej pory wszystko co zostanie zapisane do `bio`, zostanie najpierw

**Listing 6.** Funkcja `server_ssl_initialize_ctx` z pliku `ssl-server.c`

```
SSL_CTX *server_ssl_initialize_ctx() {
    SSL_CTX *ctx;
    BIO *dhbio;
    SSL_library_init();
    SSL_load_error_strings();
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv23_server_method());
    if (SSL_CTX_use_certificate_file(ctx, CERTIFICATE_FILE, SSL_FILETYPE_PEM) !=
        1) {
        fprintf(stderr, "Błąd wczytywania certyfikatu.\n");
        ERR_print_errors_fp(stderr);
        return NULL;
    }
    if (SSL_CTX_use_PrivateKey_file(ctx, PRIVATE_KEY, SSL_FILETYPE_PEM) != 1) {
        fprintf(stderr, "Błąd wczytywania klucza prywatnego.\n");
        ERR_print_errors_fp(stderr);
        return NULL;
    }
    if (SSL_CTX_check_private_key(ctx) != 1)
    {
        fprintf(stderr, "Klucz prywatny zdaje się nienależać do certyfikatu.\n");
        ERR_print_errors_fp(stderr);
        return NULL;
    }
    if ((dhbio = BIO_new_file(DH_FILE, "r")) == NULL) {
        fprintf(stderr, "Błąd we wczytywaniu parametrów DH. Pomijamy ten krok.\n");
        ERR_print_errors_fp(stderr);
    }
    else {
        DH *dh = PEM_read_bio_DHparams(dhbio, NULL, NULL, NULL);
        BIO_free(dhbio);
        if (SSL_CTX_set_tmp_dh(ctx, dh) < 0) {
            fprintf(stderr, "Błąd w ustawianiu parametrów DH. Pomijamy.\n");
            ERR_print_errors_fp(stderr);
        }
    }
}
```

**Listing 7.** Funkcja z pliku `ssl-server.c` obsługująca sygnał `SIGCHLD`

```
void sigchild_handler(int sig) {
    int status;
    pid_t pid;
    pid = wait(&status);
    if (WIFEXITED(status))
        fprintf(stderr, "Proces
        %d wrocil z
        kodem %d.\n", pid, WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        fprintf(stderr, "Proces %d został
        zabity sygnałem %d.\n", pid, WTERMSIG(status));
}
```

zamienione na ciąg `base64`, a następnie wyświetlone na standardowe wyjście.

Najszybszym sposobem na połączenie się ze zdalnym serwerem SLL/TLS jest więc stworzenie łańcucha BIO składającego się z BIO typu `BIO_s_connect()` (BIO typu `source/sink`) oraz z BIO filtrującego typu `BIO_f_ssl()`. Przeglądając

dokumentację (polecam *man bio* jako bazę), znajdziemy funkcję `BIO_new_ssl_connect()`, która zwróci nam właśnie taki łańcuch. Adres hosta i port ustalimy funkcją `BIO_set_conn_hostname()` – widać od razu jak przyjemnie pracuje się z BIO – nie musimy się nawet martwić o obsługę gniazd na niskim poziomie (co jest jedną ze szkół



pisania aplikacji z openssl). W celu uzyskania dostępu do wskaźnika na połączenie SSL znajdującego się w naszym łańcuchu BIO, używamy funkcji `BIO_get_ssl()`, co widać na Listing 4, zawierającego główną część naszego klienta.

Po otrzymaniu wskaźnika na połączenie ssl, warto jeszcze ustawić flagę `SSL_MODE_AUTO_RETRY`, dzięki której nie musimy się martwić o problem z częściowym wykonaniem operacji zapisu czy odczytu – zatroszczy się o to proto-

kół. W tym miejscu jesteśmy gotowi do zrealizowania w końcu połączenia – robi to dla nas funkcja `BIO_do_connect()`. Następną rzeczą jaką zrobimy, to zweryfikowanie certyfikatu serwera (`SSL_get_verify_result()`), oraz wyświetle-

**Listing 8.** *Ssl-server.c – główna funkcja*

```
#define DEFAULTPORT      "4433"
#define CERTIFICATE_FILE "
/home/drg/ssl/clients/cert-server.pem"
#define PRIVATE_KEY     "
/home/drg/ssl/clients/cert-server-key.pem"
#define DH_FILE         "
/home/drg/ssl/clients/dhserver.pem"
#define RESPONSE        "
HTTP/1.0 200 OK\r\nContent-type:
text/plain\r\n\r\n"
void sigchild_handler(int sig);
SSL_CTX *server_ssl_initialize_ctx();
int main(int argc, char **argv) {
    char *portbuff;
    int rv;
    BIO *listen_bio;
    SSL_CTX *ctx;
    SSL *ssl;
    char buf[1024];
    int len;
    if (!ctx = server_ssl_initialize_ctx()) {
        fprintf(stderr, "Niestety.\n");
        exit(1);
    }
    if (argc > 1)
        portbuff = strdup(argv[1]);
    else
        portbuff = strdup(DEFAULTPORT);
    signal(SIGCHLD, sigchild_handler);
    listen_bio = BIO_new_accept(portbuff);
    BIO_set_bind_mode(listen_bio,
BIO_BIND_REUSEADDR);
    if (BIO_do_accept(listen_bio) <= 0) {
        fprintf(stderr, "Błąd w
przygotowaniu do obsługi klientów.\n");
        ERR_print_errors_fp(stderr);
        exit(0);
    }
    while (1) {
        BIO *client_bio = BIO_new_ssl(ctx, 0);
        BIO *buffered_bio;
        BIO_get_ssl(client_bio, &ssl);
        if (!ssl) {
            fprintf(stderr, "Błąd ssl.\n");
            ERR_print_errors_fp(stderr);
            BIO_free_all(listen_bio);
            SSL_CTX_free(ctx);
            exit(1);
        }
        SSL_set_verify(ssl, SSL_VERIFY_
PEER | SSL_VERIFY_CLIENT_
ONCE, NULL);
        SSL_set_mode(ssl, SSL_
MODE_AUTO_RETRY);
        BIO_set_accept_bios(listen_bio, client_bio);
        if (BIO_do_accept(listen_bio)
<= 0) {
            fprintf(stderr, "Błąd w
przyjmowaniu połączenia.\n");
            ERR_print_errors_fp(stderr);
            continue;
        }
        client_bio = BIO_pop(listen_bio);
        rv = fork();
        if (rv > 0) {
            fprintf(stderr, "Mam mowego
klienta, pid=%d\n", rv);
            continue;
        }
        if (rv == -1) {
            fprintf(stderr, "Błąd w tworzeniu
nowego procesu: %s.\n", strerror(errno));
            BIO_free(client_bio);
            continue;
        }
        if (BIO_do_handshake(client_bio) <= 0) {
            fprintf(stderr, "Błąd podczas
SSL handshake.\n");
            ERR_print_errors_fp(stderr);
            BIO_free_all(client_bio);
            return 1;
        }
        buffered_bio = BIO_new(BIO_f_
buffer());
        client_bio = BIO_push(buffered_
bio, client_bio);
        BIO_puts(client_bio, RESPONSE);
        while (1) {
            len = BIO_gets(client_bio, buf, 1024);
            if (len <= 0) {
                fprintf(stderr, "Błąd w
czytaniu z gniazda.\n");
                ERR_print_errors_fp(stderr);
                break;
            }
            BIO_write(client_bio, buf, len);
            if ((buf[0] == '\r') || (buf[0] == '\n'))
                break;
        }
        BIO_puts(client_bio, "milo,
ze wpadles.\r\n");
        rv = BIO_flush(client_bio);
        BIO_free_all(client_bio);
        exit(0);
    }
}
if (portbuff)
    free(portbuff);
return 0;
}
```

nie informacji na jego temat przy pomocy mało ambitnej funkcji przedstawionej na Listingu 5.

Teraz, po upewnieniu się, że połączenie SSL/TLS jest już finalnie zestawione (`BIO_do_handshake()`), możemy rozpocząć wymianę danych – mamy do dyspozycji takie funkcje jak `BIO_read()`, `BIO_write()`, `BIO_gets()` oraz `BIO_puts()`. Wysyłamy zatem zapytanie HTTP zdefiniowane w `REQUEST`, po czym odbieramy dane i wyświetlamy je na standardowe wyjście. Na końcu wypada posprzątać łańcuch BIO za pomocą funkcji `BIO_free_all()` oraz szczęśliwie zakończyć program. Kompilacja oraz przykładowe uruchomienie programu przedstawia Rysunek 1.

## Zaserwujmy coś

Analogicznie do przypadku klienta, napiszemy funkcje `server_ssl_initialize_ctx()` inicjalizującą kontekst SSL – znajduje się ona na Listingu 6. O ile serwer nie oczekiwał od klienta certyfikatu, o tyle klient serwerowi nie popuści. Za pomocą funkcji `SSL_CTX_use_certificate_file()` i `SSL_CTX_use_PrivateKey_file()` serwer załaduje wcześniej stworzony certyfikat oraz pasujący do niego klucz prywatny. O faktyczne przypasowanie klucza prywatnego do certyfikatu zatroszczy się funkcja `SSL_CTX_check_private_key()`.

W odróżnieniu od klienta, serwer ustawia parametry dla algorytmu Diffiego-Hellmana za pomocą kombinacji funkcji `BIO_new_file()`, `PEM_read_bio_DHparams()` oraz `SSL_CTX_set_tmp_dh()`. Nasz serwer dla każdego nowego klienta będzie tworzył nowy proces potomny funkcją `fork()`, dlatego w celu uniknięcia procesów zombie, przechwytujemy sygnał `SIGCHLD`, który wysyłany jest po zakończeniu procesu potomnego:

```
signal(SIGCHLD, sigchild_handler);
```

jego obsługą zaś zajmie się funkcja `sigchild_handler` przedstawiona na Listingu 7, która wyświetli informację czy proces umarł ze starości czy też

przedwcześnie wskutek nieszczęśliwego wypadku (na przykład poprzez wysłanie do niego określonego sygnału).

Przyszedeł w końcu moment na zastosowanie BIO po stronie serwera. Funkcja `BIO_new_accept()` z podanym jako argument numeru port (w postaci tekstu) stworzy nowy łańcuch BIO, za którym schowa się obsługa sieci – znów nie trzeba wypełniać żadnych struktur adresowych czy tworzyć gniazd. Do rozpoczęcia nasłuchu służy funkcja `BIO_do_accept()`. Pierwsze jej wywołanie przygotuje BIO do nasłuchu, następne będzie już oczekiwało na nowe połączenia. Każde nowo zaakceptowane połączenie trafi do BIO łańcucha o nazwie `listen_bio`. Konstrukcja

```
client_bio = BIO_pop(listen_bio);
```

zdejmie klienta z łańcucha `listen_bio`, dzięki czemu serwer będzie mógł przyjmować nowe połączenia, a do obsługi nowo przyjętego stworzy proces potomny. Po pomyslnym zestawieniu połączenia (`BIO_do_handshake()`), proces potomny stworzy doda do łańcucha BIO `client_bio` BIO filtrujące `BIO_f_buffer()`, które zapewni buforowane wejście i wyjście w komunikacji z klientem. Następnie serwer wysła ślepo klientowi przykładową odpowiedź HTTP, po czym odczytuje dane od klienta (zapytanie HTTP), odsyła mu je i na końcu dodaje coś od siebie. Na Rysunku 2 widać kompilację i działanie serwera, który dobrze obsługuje zarówno naszego własnoręcznie zbudowanego klienta, jak i popularną przeglądarkę WWW. Listing 8 natomiast zawiera źródła z sercem naszego demona.

## Podsumowanie

Programowanie z biblioteką `Openssl` nie jest wcale tak trudne, jak by się mogło wydawać. Wprawdzie przedstawione w artykule przykłady są bardzo proste, jednak widać, że – zwłaszcza dzięki abstrakcyjnym BIO – w łatwy sposób można zestawić szyfrowane połączenie. ●