

Secure Sockets Layer

Marcin Dąbrowski

Krzysztof Luks

3 grudnia 2000 roku

Spis treści

1	Wstęp	3
2	Wprowadzenie do protokołu SSL	4
3	Założenia	5
4	Budowa protokołu SSL	6
5	OpenSSL	8
5.1	Kompilacja	8
5.2	Program openssl	9
5.2.1	Przykłady	10
5.2.2	Tworzenie certyfikatu dla serwera	10
6	Połączenie i transmisja	11
6.1	SSL Record Layer	12
6.2	Zmiana specyfikacji kodowania	13
6.3	Protokół ostrzeżeń	13
6.4	SSL Handshake Protocol	13
7	Programowanie	16
7.1	Nawiązywanie sesji	16
7.2	Kompilacja	16
7.3	Klient	17
7.4	Serwer	18
8	Tunelowanie	22
8.1	stunnel	22
9	Programy używające SSL	24
9.1	mod_ssl	24
9.2	Przeglądarki	24
9.3	POP, IMAP itp.	25
10	Bibliografia	26

1 Wstęp

Z uwagi na znikomą ilość materiałów dotyczących SSL'a w języku polskim, zdecydowaliśmy się nie spolszczać większości z używanych pojęć. Ma to na celu ułatwienie wyszukiwania dalszych informacji na ten temat. Część danych (specyfikacje i inne dokumenty, także i ten) dostępne będą pod adresem: <http://lilo.pjwstk.waw.pl/projekty/ssl/>.

2 Wprowadzenie do protokołu SSL

Protokół SSL został zaprojektowany z myślą dostarczenia praktycznego, zorientowanego na warstwę aplikacji [1], opartego na połączeniu mechanizmu dla komunikacji typu klient-serwer w Internecie. Dynamiczny rozwój Internetu, jak i samej technologii WWW, przyniósł ze sobą potrzebę bezpiecznej ochrony danych przesyłanych otwartymi kanałami sieci. Wcześniejszy protokół SSL w wersji 2.0 stał się *de facto* standardem dla kryptograficznej ochrony połączeń HTTP w sieci. Ale miał on swoje ograniczenia – zarówno w warstwie zabezpieczeń, jak i w funkcjonalności – tak więc została podniesiona jego jakość, dodano wiele nowych rozszerzeń; powstał SSL wersji 3.0. SSL 2.0 miał wiele słabości pod względem bezpieczeństwa, które SSL 3.0 stara się naprawić. Więcej na ten temat w dokumencie „Analysis of the SSL 3.0 Protocol” [2]. Tam też znajduje się analiza techniczna siły kryptograficznej protokołu SSL 3.0.

Będący w opracowaniu TLS [3] także bazuje na protokole SSL. OpenSSL posiada wsparcie dla TLS1.0, ale nie będziemy się tu nim zajmować (TLS'em), ponieważ nie jest on zbyt szeroko używany.

Podsumowując, SSL 3.0 powstał, mając na celu dostarczenie internetowym aplikacjom klient-serwer praktycznego, mającego szerokie zastosowanie mechanizmu komunikacji.

3 Założenia

Twórcy protokołu założyli sobie następujące cele (w kolejności ich ważności):

Bezpieczeństwo kryptograficzne SSL powinien być używany do ustanowienia bezpiecznego połączenia pomiędzy dwoma interesującymi nas końcami.

Niezależność Niezależni od siebie programiści powinni mieć możliwość dostarczania aplikacji opartych o SSL 3.0, które będą miały możliwość pomyślnej wymiany kryptograficznych parametrów bez potrzeby znajomości kodu innych aplikacji. (Chodzi tu o SSL Handshake Protocol, który umożliwia wymianę kluczy, zapoznanie się aplikacji ze sobą oraz ustalenie tajnego klucza sesji w ustandaryzowany sposób.)

Rozszerzalność SSL powinien być zbudowany niczym ramka, w którą można wkomponowywać nowe metody kodowania. Ma to dwa znaczenia: zapobiec tworzeniu nowych protokołów (i ryzykować potencjalne słabości nowego produktu), oraz zapobiec potrzebie implementacji całkowicie nowej biblioteki.

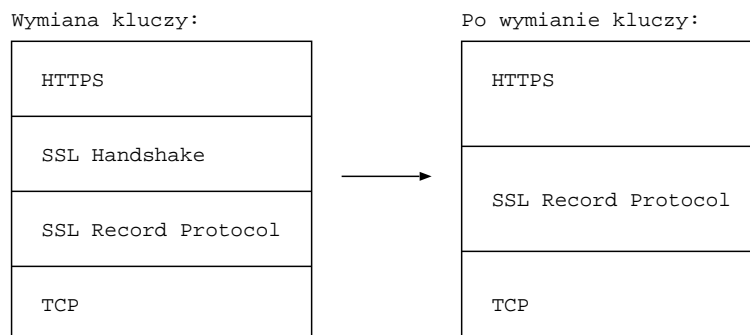
Podobna skuteczność Operacje kryptograficzne znane są ze swoich wymagań względem CPU, szczególnie operacje na kluczach publicznych. Z tego też powodu SSL wdrożył opcjonalny system pamięci podręcznej mający na celu zmniejszenie liczby połączeń, które musiałyby zaczynać się od początku. To powinno zredukować także ruch w sieci (powodowany potrzebą wymiany kluczy i ustalania atrybutów połączenia).

4 Budowa protokołu SSL

Podstawowym celem protokołu SSL jest zapewnienie prywatności i zaufania pomiędzy dwiema komunikującymi się aplikacjami. Protokół złożony jest z dwu warstw: pierwsza, ułożona na niższym poziomie, oparta na jakimś zaufanym protokole transportu, w naszym wypadku będzie to zapewne TCP, to SSL Record Protocol. Jest on używany do przenoszenia (*encapsulate*) różnych wyższych protokołów. SSL Record Layer zapewnia poufność, autentyczność (przenoszonych danych) i ochronę odpowiedzi ponad zaufanym protokołem transportu, takim jak TCP.

Warstwę wyższą stanowi protokół SSL Handshake, czyli protokół wymiany kluczy który inicjalizuje i synchronizuje kryptograficzne stany obu łączących się końcówek, czyli na przykład Netscape'a łączącego się z Apachem-SSL (na porcie 443). Po zakończeniu wymiany kluczy, wrażliwe dane aplikacji mogą być transmitowane przez SSL Record Layer. SSL Handshake umożliwia klientowi i serwerowi „zapoznać się” ze sobą, ustalić sposób/algorytm kodowania transmisji, oraz wymienić klucze używane do tej wymiany. To wszystko dzieje się jeszcze zanim pierwsze bity danych przepłyną warstwą aplikacji [1].

Tak więc, SSL podzielony jest na dwie warstwy, z których każda korzysta z usług dostarczanych przed warstwę niższą, a sama dostarcza funkcjonalności dla warstw wyższych.



Rysunek 1: Położenie protokołu SSL w warstwowym modelu sieci

Podstawową zaletą SSL'a jest to, iż jest on niezależny od protokołów warstwy aplikacji. Wygląda to tak, jakby dla aplikacji protokół SSL był przezroczysty. SSL zapewnia bezpieczne połączenie, które ma trzy podstawowe właściwości:

-
- połączenie jest prywatne. Cały czas po inicjującym połączeniu używana jest silna kryptografia, między innymi do ustalenia sekretnego klucza sesji.
 - tożsamość obydwu stron może być autoryzowana za pomocą kryptografii asymetrycznej, lub z kluczem publicznym. (RSA, DSS, etc.).
 - połączenie jest 100% pewne. Transport wiadomości zawiera sprawdzanie integralności wiadomości przy pomocy zakodowanego MAC. Do obliczenia MAC używane są funkcje haszujące (np. SHA, MD5, etc.).

5 OpenSSL

OpenSSL jest zbiorem bibliotek implementujących *Secure Sockets Layer* (SSL v2/v3) i *Transport Layer Security* (TLS v1) a także niezbędne funkcje pomocnicze. Została stworzona w oparciu o SSLeay autorstwa Erica Younga i Tima Hudsona. Początki SSLeay sięgają roku 1995, kiedy to Eric rozpoczął prace nad swoją biblioteką. Ostatnią wersja SSLeay nosiła numer 0.9.0 i ukazała się w kwietniu 1998. Latem tego roku rozpoczęły się prace nad OpenSSL, które jako podstawę wykorzystały nieopublikowaną wersję SSLeay (0.9.1b). Implementacja OpenSSL rozwijana jest do dziś. W jej skład wchodzi następujące biblioteki:

libssl podstawowe funkcje implementujące standardy SSLv2, SSLv3, TLSv1 oraz kod pozwalający na użycie SSLv2, SSLv3 i TLSv1 w jednym programie.

libcrypto algorytmy szyfrujące oraz X.509 v1/v3 wykorzystywane przez SSL i TLS ale nie będące ich ścisłą częścią. libcrypto zawiera:

- Algorytmy szyfrujące: DES, RC2, RC4, Blowfish, Idea
- Funkcje skrótów: MD2, MD5, SHA, SHA-1, MDC2
- Algorytmy szyfrujące z kluczem publicznym: RSA, DSA, Diffie-Hellman
- Certyfikaty X.509
- Inne: kodowanie base64, BIO (Basic Input/Output), funkcje zarządzające i inne pomocnicze.

openssl Tekstowy interface do funkcji bibliotecznych. Implementuje podstawowego klienta, serwer, funkcje do generowania, podpisywania i weryfikacji certyfikatów.

5.1 Kompilacja

Instalacja i konfiguracja omówiona zostanie na przykładzie OpenSSL w wersji 0.9.4. Źródła dostępne są w Internecie pod adresem <http://www.openssl.org>. Sama instalacja przebiega w standardowy sposób czyli:

```
$ ./config
$ make
$ make test
$ make install
```


Skrypt `config` próbuje automatycznie rozpoznać typ systemu i kompilatora i odpowiednio ustawia zmienne używane przy kompilacji i późniejszej instalacji. Jeżeli z jakiś powodów automatyczna konfiguracja nie powiedzie się można użyć polecenia `Configure` podając jako argument typ systemu np. `linux-elf`. Standardowo OpenSSL rozpoznaje m. in. następujące systemy:

- AIX
- *BSD (Open, Net, Free, BSDI)
- HPUX
- IRIX
- Linux
- NeXT
- SCO
- Solaris
- Win32
- inne

Dodatkowo skryptowi `config` można przekazać kilka opcji wpływających na późniejsze możliwości skompilowanego pakietu:

no-threads kompilacja bez wsparcia dla aplikacji wielowątkowych

threads kompilacja ze wsparciem dla wielowątkowości

no-asm nie zostaną użyte procedury napisane w assemblerze

no-*algorytm* kompilacja z wyłączeniem podanego algorytmu szyfrowania.

Np.: `config no-rc5 no-idea` wyłączy algorytmy objęte patentami.

5.2 Program openssl

Po zakończonej kompilacji oprócz bibliotek dostajemy narzędzie o nazwie `openssl` będące interface'em do większości funkcji biblioteki `crypto` stanowiącej część pakietu OpenSSL. Oprócz tego polecenie `openssl` implementuje podstawowego klienta i serwer SSL.

5.2.1 Przykłady

```
OpenSSL> md5
Tak właśnie wygląda generowanie skrótów 'md5'
w trybie poleceń.
^D
7370deac2060aadb525892de7a8ee13c

$cat > /tmp/test
Ta wiadomość jest tajna.
Nie czytać!
^D
$openssl
OpenSSL> idea -in /tmp/test -out /tmp/test1
enter idea-cbc encryption password:
Verifying password - enter idea-cbc encryption
password:
OpenSSL> idea -in /tmp/test1 -d
enter idea-cbc decryption password:
Ta wiadomość jest tajna.
Nie czytać!
```

5.2.2 Tworzenie certyfikatu dla serwera

Najpierw należy wygenerować prywatny klucz RSA (zaszyfrowany 3DES'em):

```
$ openssl genrsa -des3 -out server.key 1024
```

Dla bezpieczeństwa klucz jest zabezpieczony za pomocą tzw. *passphrase*. Można zrezygnować z szyfrowania klucza, ale wtedy każdy kto wejdzie w jego posiadanie będzie mógł się pod nas podszywać.

Następnie należy wygenerować prośbę o podpis certyfikatu (*Certificate Signing Request – CSR*).

```
$ openssl req -new -key server.key -out server.csr
```

Mając tak wygenerowany *CSR* możemy zwrócić się do którejś w firm zajmujących się podpisywaniem certyfikatów lub podpisać go samemu. Oczywiście w tym drugim przypadku mało kto zechce nam zaufać – taki certyfikat będzie miał bardzo ograniczone zastosowanie.

6 Połączenie i transmisja

SSL jest protokołem warstwowym. Na każdej warstwie przesyłka może składać się z bloków określających: długość, opis oraz zawartość. SSL pobiera wiadomości do transmisji, fragmentuje je na mniejsze, łatwiejsze do zarządzania bloki, opcjonalnie kompresuje je, daje MAC, koduje i transmituje rezultat. Otrzymane dane są dekodowane, weryfikowane, dekompresowane i składane z powrotem w całość, a potem dostarczane do klienta operującego na wyższej warstwie.

Protokół SSL opiera się na stanach. Odpowiedzialnością protokołu SSL Handshake jest koordynacja stanów klienta i serwera, w taki sposób by mogły się ze sobą bez przeszkód komunikować. Stany są reprezentowane podwójnie, jako aktualny stan, lub jako stan oczekujący (nierozstrzygnięty). SSL może utrzymywać kilka niezależnych bezpiecznych połączeń; dodatkowo, może utrzymywać kilka niezależnych sesji.

Stan sesji opisują następujące elementy:

identyfikator sesji sekwencja bajtów wybrana przez serwer do zidentyfikowania aktywnej, lub możliwej do dokończenia sesji.

certyfiakat certyfiakat X509.v3. Może być zerem.

metoda kompresji algorytm używany do kompresji wiadomości, odpowiedni dla wybranej metody kodowania.

algorytm definiuje algorytm kodowania danych (null, DES, etc.) oraz algorytm funkcji haszującej MAC (MD5, SHA). Dodatkowo definiuje takie atrybuty jak np. długość hasza. (hash-size).

odnawialność jest to flaga, która definiuje, czy sesja może być użyta do zainicjowania nowego połączenia.

Stan połączenia definiują następujące elementy:

liczba losowa wybrana losowa liczba, inna dla klienta i dla serwera, inna dla każdego połączenia.

serwer MAC sekret klucz używany w funkcji haszującej MAC dla danych wysyłanych przez serwer.

klient MAC sekret klucz używany w funkcji haszującej MAC dla danych wysyłanych przez klienta.

klucz serwera klucz używany do kodowania danych wysyłanych przez serwer.

klucz klienta klucz używany do kodowania danych wysyłanych przez klienta.

wektory inicjalizacji jeśli używany jest algorytm kodowania w stanie CBC, przechowywane są tutaj wektory inicjalizacji dla każdego klucza. To pole jest inicjalizowane przez SSL Handshake Protocol.

numery sekwencyjne każda końcówka połączenia utrzymuje swoją własną pulę numerów dla wysyłanych i odbieranych danych. Gdy zostanie wysłana/otrzymana wiadomość o zmianie kodowania, odpowiednie numery sekwencyjne są zerowane.

6.1 SSL Record Layer

Ta warstwa otrzymuje dane od warstwy wyższej i ma za zadanie zająć się ich bezpiecznym przetransmitowaniem kodowanym kanałem na drugi koniec połączenia. Otrzymane porcje danych są fragmentowane na mniejsze części. Taki pakiet wkładany jest do struktury `_SSLPlaintext_`. Potem następuje kompresja z użyciem protokołu określonego parametrami stanu sesji. Zwykle jest to zdefiniowane jako 'null', jeżeli nie było ustawiane inaczej. Algorytm kompresji niejako tłumaczy strukturę `_SSLPlaintext_` w strukturę `_SSLCompressed_`. Kompresja **musi** być bezstratna i nie może powiększać długości pakietu o więcej niż 1024 bajty. Cała wiadomość jest chroniona za pomocą kodowania oraz MAC (jest to jakby suma kontrolna przekazu, *Message Authentication Code*), ustalonych w parametrach stanu sesji. Kodowanie i operacje MAC tłumaczą strukturę `_SSLCompressed_` w strukturę `_SSLCiphertext_`. Cały pakiet zaopatrzony zostaje w odpowiedni numer sekwencji, co zapobiega zgubieniu, przekłamaniu lub nadwyżce wiadomości. Należy zauważyć, że MAC obliczany jest przed kodowaniem. Kodowany jest cały blok danych, włączając w to obliczony przed chwilą MAC. Funkcja generująca MAC:

```
hash(MAC_write_secret + pad2 +  
      hash(MAC_write_secret + pad1 + seq_num +  
            SSLCompressed.type + SSLCompressed.length +  
            SSLCompressed.fragment));
```

gdzie:

pad1 0x36 powtórzone 48 razy dla MD5, lub 40 razy dla SHA

pad2 0x5c powtórzone 48 razy dla MD5, lub 40 razy dla SHA

seq_num numer sekwencyjny tej wiadomości

hash algorytm zdefiniowany w atrybucie stanu sesji

6.2 Zmiana specyfikacji kodowania

Podczas sesji możliwe jest zmienienie algorytmu kodowania. Jedna ze stron połączenia wysyła specjalny pakiet złożony z jednej zakodowanej i skompresowanej wiadomości. Jest to dokładnie pojedynczy bajt o wartości 1. (SSL jest tak bezpieczny, że potrafi skompresować i zakodować nawet pojedynczą jedynekę :)). Reakcją jest wysłanie takiej samej wiadomości jako potwierdzenia. Obie końcówki zamieniają stany oczekujące na aktualne, wymieniają uścisk dłoni (handshake), klucze i certyfikaty i odsyłają odpowiednie informacje do drugiego końca. Wystąpienie takiej sytuacji w nieprzewidzianym momencie powoduje wygenerowanie ostrzeżenia.

6.3 Protokół ostrzeżeń

Warstwa SSL Record wspomaga obsługę błędów. Wiadomości będące ostrzeżeniem cechują się typem (fatalny, zwykły...) oraz opisem. Jeżeli ostrzeżenie jest typu 'fatal' powoduje to natychmiastowe zerwanie połączenia. Tak jak inne wiadomości, także i te są kodowane i kompresowane. Ostrzeżenia są następujące:

kończące się połączenie zarówno klient jak i serwer muszą wiedzieć, że połączenie się kończy, co ma zapobiec pewnym typom ataków. Po otrzymaniu takiego pakietu, każdy następny idący tym połączeniem jest ignorowany.

błąd jeżeli któraś końcówka wykryje błąd, to wysyła właśnie takie ostrzeżenie na drugi koniec połączenia. Jeżeli był to błąd klasy *fatal*, połączenie jest natychmiast zamykane.

6.4 SSL Handshake Protocol

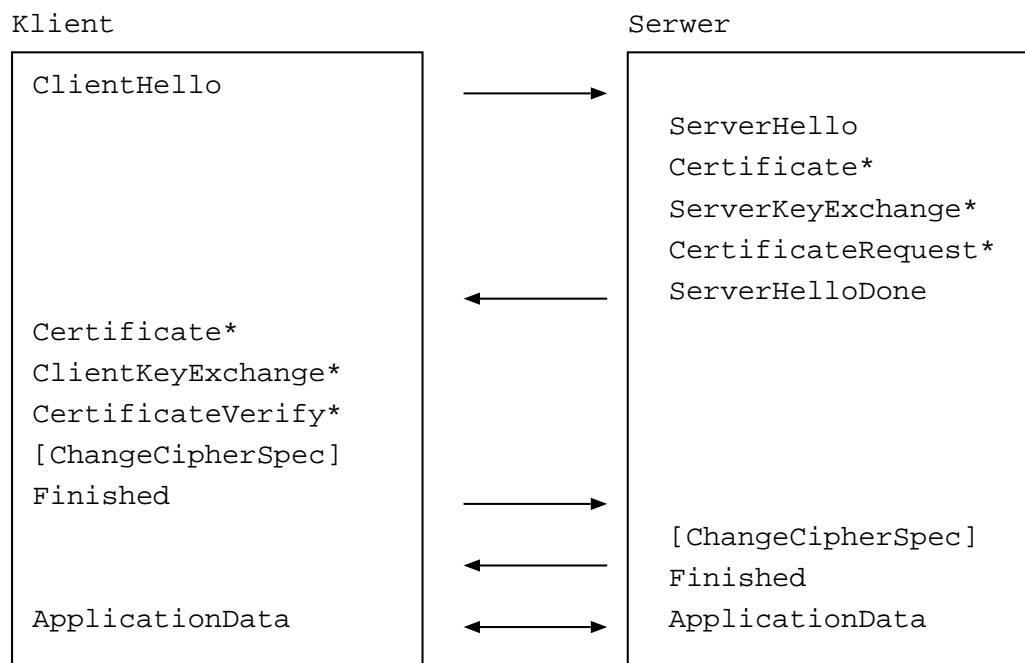
Parametry kryptograficzne stanu sesji są produkowane właśnie przez ten protokół, który działa ponad warstwą SSL Record. Gdy klient i serwer łączą się po raz pierwszy, porozumiewają się odnośnie wersji protokołu, wybierają algorytm kodujący, opcjonalnie rozpoznają się oraz używają technik 'klucza publicznego' do ustalenia/wygenerowania klucza sesji. Te procesy są wykonywane właśnie przez protokół SSL Handshake, i mogą być streszczone następująco: Klient wysyła do serwera wiadomość `client_hello`, na którą to serwer musi odpowiedzieć wiadomością `server_hello`, inaczej wystąpi błąd i połączenie zostanie zerwane. Obydwie wiadomości `_hello` są używane do ustalenia: wersji protokołu, ID sesji, zestawu algorytmów kodowania oraz

metod kompresji. Dodatkowo generowane i wymieniane są dwie wartości losowe: `clienthello.random` oraz `serverhello.random`. Po wysłaniu `_hello` serwer wysyła swój certyfikat w celach autoryzacji. Dodatkowo, możliwa jest wymiana kluczy (jeśli serwer nie ma certyfikatu, lub jest on tylko do podpisywania). Jeżeli serwer zostanie zautoryzowany domaga się certyfikatu od klienta, jeśli to konieczne, odpowiedniego dla zadanego algorytmu kodowania. Serwer wysyła wtedy także wiadomość `server_hello_done`, która oznacza, że zakończył już fazę `_hello`. Teraz serwer czeka na odpowiedź klienta.

Jeżeli serwer wysłał żądanie o certyfikat, to klient musi wysłać teraz swój, lub jeśli takowego nie posiada, ostrzeżenie `no_certificate` (nie jest typu *fatal*). Potem klient wysyła wiadomość o wymianie kluczy, zależnie od wybranego algorytmu kodowania. Jeśli klient wysłał swój certyfikat, z możliwością podpisywania, to dodatkowo wysyłana jest podpisana wiadomość weryfikująca ten certyfikat. W tym momencie klient wysyła wiadomość o wymianie algorytmu kodowania, co sprawia, że ustalony oczekujący algorytm staje się aktualnym. Od razu po tym wysyła wiadomość o zakończeniu wymiany poglądów z serwerem (*Finished*), używając ustalonego algorytmu, kluczy publicznych i prywatnych. W odpowiedzi serwer wysyła swoją wiadomość o wymianie algorytmu (czyli ten ustalany staje się aktualnym) i wysyła swoją wiadomość o zakończeniu negocjacji. W tym momencie SSL Handshake kończy pracę; klient i serwer mają już możliwość wymiany danych aplikacji.

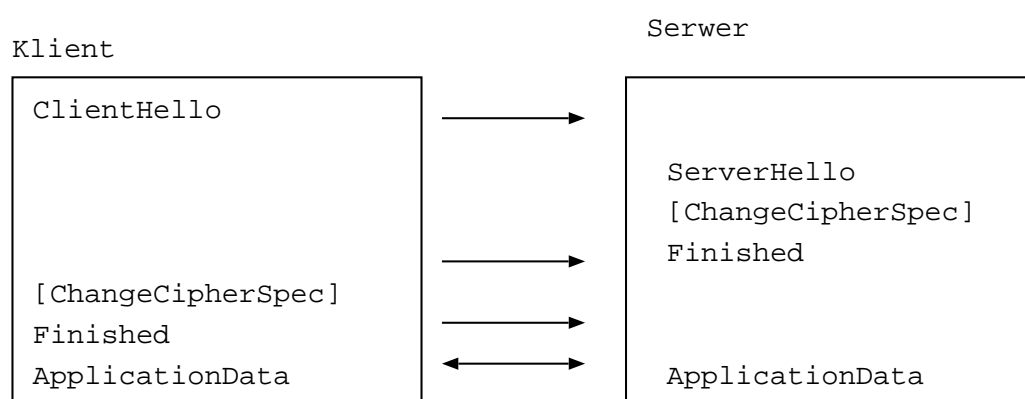
Jeżeli klient i serwer zdecydują się odnowić poprzednią sesję, lub zduplikować bieżącą (zamiast negocjować nowe parametry od początku), wymiana informacji wygląda jak następuje: Klient wysyła `client_hello` używając ID Sesji która ma być odnowiona. Serwer sprawdza w pamięci podręcznej, czy ma takie ID. Jeżeli znajdzie i jest chętny do odnowienia połączenia na warunkach z tego stanu sesji, wysyła `server_hello` z tym samym ID Sesji. A tym momencie zarówno klient, jak i serwer, muszą wysłać wiadomość o zmianie algorytmu i przejść od razu do wysłania wiadomości o zakończeniu negocjacji. Kiedy już połączenie jest odnowione, klient i serwer mogą już wymieniać dane aplikacji. Jeżeli natomiast podane przez klienta ID Sesji nie zostanie znalezione w pamięci serwera, to serwer generuje nowe ID Sesji i następuje pełna negocjacja parametrów (SSL Handshake w wersji full).

Należy nadmienić, że wszystkie wiadomości podczas negocjacji muszą być wysyłane w odpowiedniej kolejności, inaczej powitanie błąd klasy *fatal* co spowoduje zerwanie połączenia.



Rysunek 2: Nawiązanie połączenia

Gwiazdka oznacza opcjonalne, lub zależne od sytuacji wiadomości, które nie zawsze są wysyłane.



Rysunek 3: Odnawianie połączenia

7 Programowanie

7.1 Nawiązywanie sesji

Informacje dotyczące sesji SSL przechowywane są w strukturze o nazwie `SSL_CTX` (kontekst połączenia SSL). Przechowywane są tam informacje o używanych w czasie sesji algorytmach szyfrowania, certyfikatach a także identyfikatory poszczególnych sesji. Identyfikatory te mogą być później ponownie użyte (do ponownego otwarcia sesji z tym samym identyfikatorem). W większości przypadków jeden program używa tylko jednego kontekstu.

```
SSL_CTX *SSL_CTX_new(void ); // Utworzenie kontekstu
void SSL_CTX_free(SSL_CTX *); // Zniszczenie kontekstu
```

Po każdym udanym nawiązaniu połączenia (funkcja `SSL_accept()`) w strukturze `SSL_CTX` zapamiętywany jest jej identyfikator. Możliwe jest również dodawanie certyfikatów klientów do cache'u w kontekście. Domyślnie rozmiar cache'a ustawiony jest na 255 pozycji.

7.2 Kompilacja

Przy pisaniu programów używających OpenSSL do dyspozycji mamy m. in. następujące nagłówki:

ssl.h Zawiera główne funkcje API SSL/TSL. Zawiera w sobie dyrektywy `include` dla `ssl2.h`, `ssl23.h`, `ssl3.h`, `tsl1.h` oraz `bio.h`, `crypto.h` i `x509.h`.

rsa.h Nagłówki funkcji związanych z algorytmem RSA. Podobnie mamy: `md2.h`, `md5.h`, `sha.h`, `idea.h` etc.

bio.h Funkcje związane z operacjami wejścia/wyjścia. Stanowią one znaczne ułatwienie przy pisaniu programów komunikujących się przez sieć na różnych platformach. Zawiera funkcje odpowiedzialne za: tworzenie i wykorzystywanie deskryptorów plików i gniazd, buforowanie operacji I/O w pamięci, funkcje filtrujące pozwalające na przezroczyste kodowanie i/lub szyfrowanie przesyłanych danych.

asn1.h Funkcje operujące na danych w formacie ASN.1.

bn.h Funkcje do operowanie na dużych liczbach.

rand.h Generacja liczb losowych.

W przykładowych programach używałem:


```
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>

gcc -o program program.c -lcrypto -lssl
```

7.3 Klient

Przykładowy program, który łączy się z serwerem na określonym porcie, odbiera od niego komunikat i wypisuje go na ekran.

```
#define PORT 31337

int main(void)
{
    int sock;
    char rc[13];
    struct sockaddr_in server;

    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    inet_aton("127.0.0.1", &server.sin_addr);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    connect(sock, (struct sockaddr *) &server,
            sizeof(struct sockaddr_in));
    recv(sock, &rc, 13, 0);
    printf("%s\n", rc);
    close(sock);
    exit(0);
}
```

To samo z użyciem SSL. Dodatkowo klient sprawdza certyfikat serwera.

```
int main(void)
{
    int sock;
    char rc[13];
    struct sockaddr_in server;
    SSL_CTX *ctx;
    SSL *ssl;
    X509 *server_cert;
    char *tmp;
    SSL_METHOD *meth;
```

```

    SSLey_add_ssl_algorithms();
    meth = SSLv2_client_method();
    ctx = SSL_CTX_new(meth);
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    inet_aton("127.0.0.1", &server.sin_addr);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    connect(sock, (struct sockaddr *)&server,
            sizeof(struct sockaddr_in));

    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, sock);
    SSL_connect(ssl);

    printf("Algorytm: %s\n", SSL_get_cipher(ssl));
    server_cert = SSL_get_peer_certificate(ssl);
    tmp =
        X509_NAME_oneline(X509_get_subject_name
                          (server_cert), 0, 0);
    printf("Podmiot: %s\n", tmp);
    tmp =
        X509_NAME_oneline(X509_get_issuer_name
                          (server_cert), 0, 0);
    printf("Wystawca: %s\n", tmp);
    X509_free(server_cert);

    SSL_read(ssl, rc, 13);

    printf("%s\n", rc);
    close(sock);
    SSL_free(ssl);
    SSL_CTX_free(ctx);
    exit(0);
}

```

7.4 Serwer

Przykładowy serwer, który oczekuje na połączenie od klienta a następnie wysyła mu komunikat.

```
#define PORT 31337
```

```
int main(void)
```

```
{
    int sock, newsock;
    char c;
    char mesg[13] = "I hear you!\n";
    struct sockaddr_in server;

    server.sin_family = AF_INET;

    server.sin_port = htons(PORT);
    server.sin_addr.s_addr = INADDR_ANY;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    bind(sock, (struct sockaddr *)&server,
          sizeof(struct sockaddr_in));
    listen(sock, 5);

    newsock = accept(sock, NULL, NULL);

    send(newsock, &mesg, 13, 0);
    close(newsock);
    shutdown(sock, 2);
    exit(0);
}
```

Jak wyżej z użyciem SSL.

```
#define PORT 31337
#define KEY "server.key"

int main(void)
{
    int sock, newsock;
    char c;
    char mesg[13] = "I hear you!\n";
    struct sockaddr_in server;
    SSL_CTX *ctx;
    SSL *ssl;
    X509 *client_cert;
    SSL_METHOD *meth;
    char *tmp;

    SSL_load_all_crypto_library();
    meth = SSLv23_server_method();
    ctx = SSL_CTX_new(meth);
    SSL_CTX_use_certificate_file(ctx, KEY,
```



```
    close ( sock );  
    SSL_free ( ssl );  
    SSL_CTX_free ( ctx );  
    exit ( 0 );  
}
```

8 Tunelowanie

Po ustanowieniu połączenia SSL działa w sposób przezroczysty dla aplikacji. Można to wykorzystać do tunelowania połączeń tak, aby zapewnić bezpieczeństwo i integralność transmisji danych dla „niebezpiecznych” protokołów takich jak POP-2, POP-3, IMAP, NNTP, SMTP i HTTP.

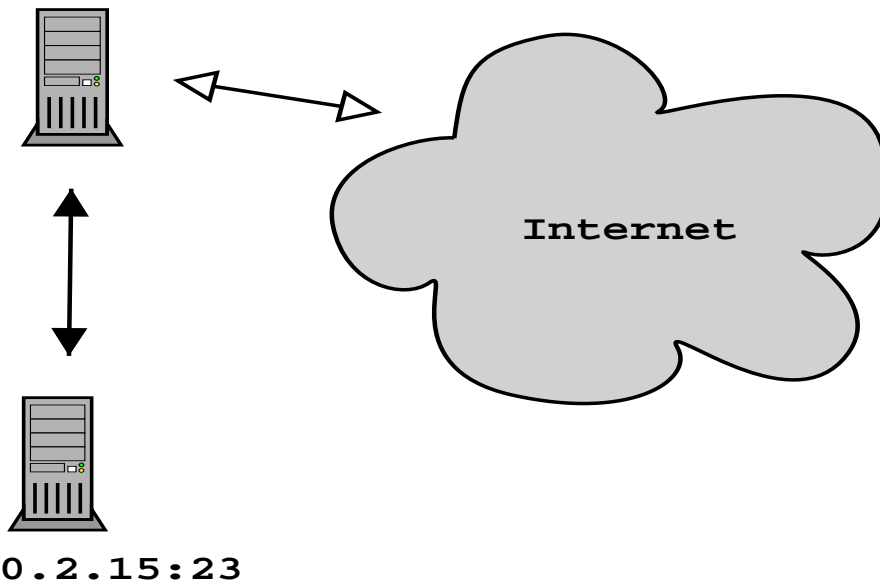
8.1 stunnel

Jednym z najpopularniejszych programów do tego celu jest stunnel. Pozwala on np. na tworzenie na Linuksie tzw. *transparent proxies* czyli takiego skonfigurowania firewalla, żeby wszystkie połączenia do zadanych hostów na zadane porty były przekierowywane na odpowiedni port na komputerze z firewallem.

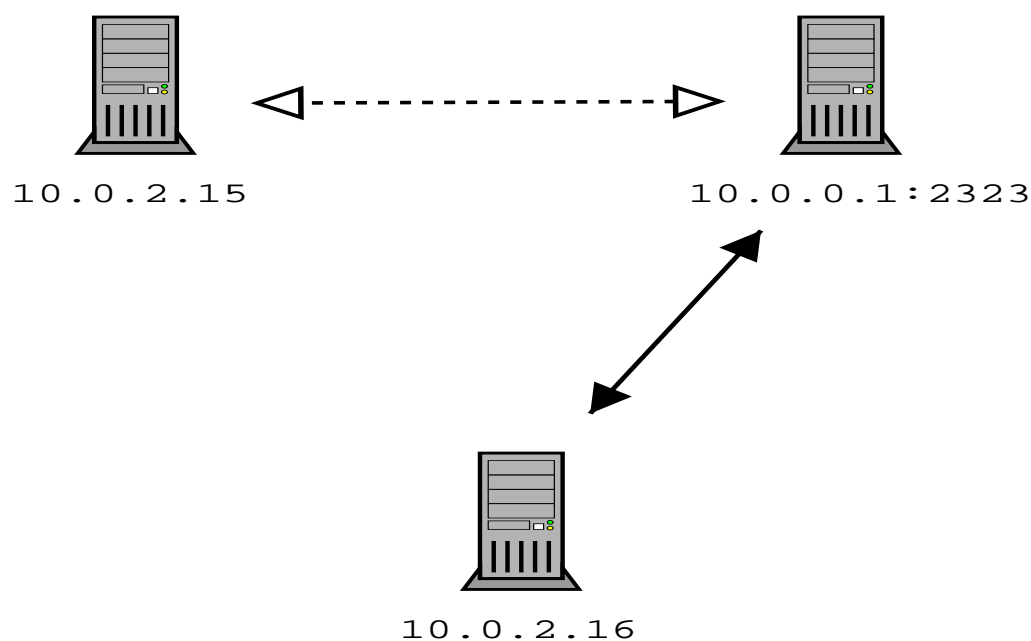
Na przykład: mamy sieć lokalną 10.x.x.x (maska 255.0.0.0. Na maszynie o adresie 10.0.0.1 jest firewall (widoczny z zewnątrz jako 123.123.123.123), który stanowi jednocześnie bramę dla wszystkich komputerów w sieci lokalnej. Chcemy aby wszystkie połączenia do firewalla na port 2323 były przekierowywane do komputera 10.0.2.15 na port 23. Odpowiednie wywołanie stunnela:

```
stunnel --transproxy 10.0.2.15 23 2323
```

123.12.1.0:2323



Rysunek 4: Połączenie z sieci zewnętrznej



Rysunek 5: Połączenie z sieci lokalnej

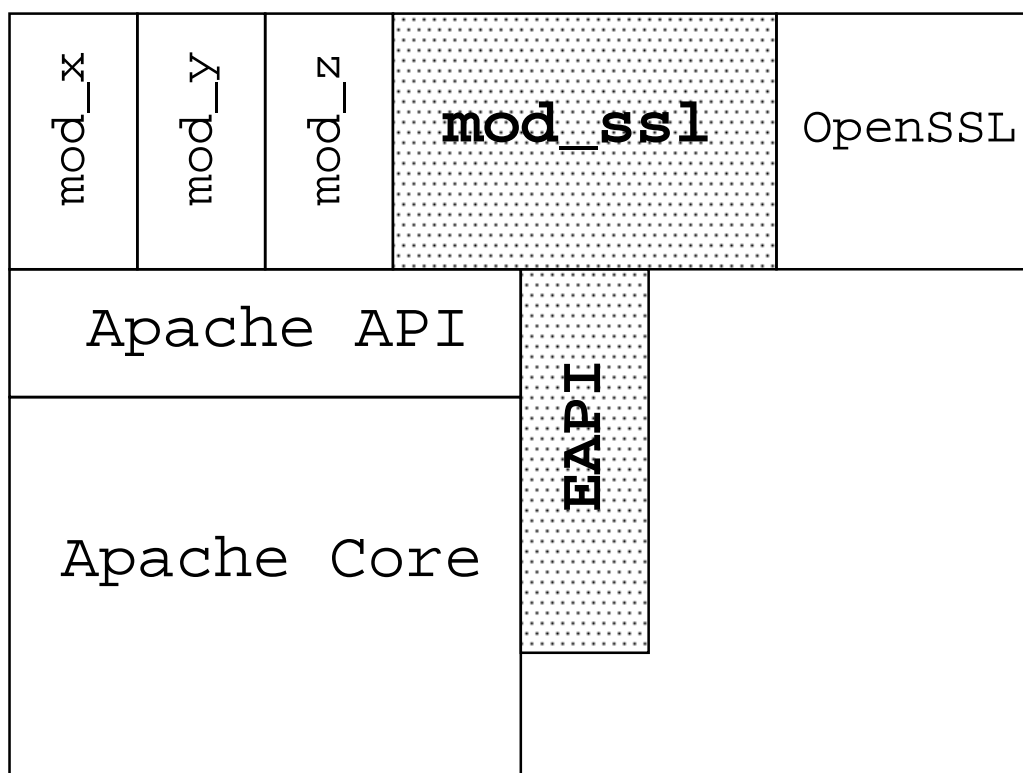
W drugim przypadku połączenie nie powiedzie się. Komputer `10.0.2.15` będzie oczekiwał danych od „prawdziwego” `10.0.2.16`, podczas gdy połączenie pochodzi od gateway’a. W pierwszym przypadku ponieważ połączenie nawiązał komputer z zewnątrz `10.0.2.15` spodziewa się pakietów z gateway’a.

Podobną funkcję jak stunnel spełnia program `stunnel`. Zaletą obydwu tych programów jest ich wieloplatformowość.

9 Programy używające SSL

9.1 mod_ssl

Jest to dynamicznie ładowany moduł do Apache'a pozwalający na używanie SSL. Został stworzony w oparciu o OpenSSL. `mod_ssl` składa się z dwóch części: właściwego modułu i *patch*'a rozszerzającego API Apache'a (*Extended API*). `mod_ssl` udostępnia pełen zestaw funkcji do nawiązywania połączeń z użyciem SSL, uwierzytelniania i zarządzania certyfikatami.



Rysunek 6: Architektura `mod_ssl`

9.2 Przeglądarki

SSL jest obsługiwane przez większość popularnych przeglądarek: Netscape, Internet Explorer, Lynx, Links, w3m ...

Należy jednak zaznaczyć, że „międzynarodowe” (nie-amerykańskie) wersje IE mają kłopoty z kluczami dłuższymi niż 512 bitów.

9.3 POP, IMAP itp.

Oprócz tego istnieją zmodyfikowane wersje wielu popularnych programów sieciowych wspierające SSL. Należą do nich serwery i klienci POP3 i IMAP, wrappery pozwalające na dodanie obsługi SSL do usług uruchamianych za pomocą `inetd`. Jest także telnet zawierający obsługę SSL.

10 Bibliografia

Bibliografia

- [1] W. Richard Stevens: *Biblia TCP/IP, tom 1, protokoły*, Wydawnictwo RM, Warszawa 1998
- [2] David Wagner, Bruce Schneier: *Analysis of the SSL 3.0 Protocol*, wersja „revised” z 15 Kwietnia 1997
- [3] Alan O. Freier, Philip Karlton, Paul C. Kocher: *The SSL Protocol, Version 3.0. Internet-Draft, IETF*, 18 Listopada 1996