Programowanie Aplikacji Sieciowych - Laboratorium 14

Bezpieczne gniazda TLS/SSL (Python - moduł `ssl`)

- Moduł `ssl` w Pythonie umożliwia wykorzystanie w aplikacji gniazd zabezpieczonych za pomocą TLS/SSL (tzw. Secure Sockets Layer). Połączenia wykorzystujące bezpieczne gniazda umożliwiają szyfrowanie połączeń oraz autentykację zarówno strony serwera jak i klienta.
- Moduł `ssl` wykorzystuje bibliotekę OpenSSL
- Moduł `ssl` dostarcza klasę, `ssl.SSLSocket`, która dziedziczy po klasie `socket.socket` i wprowadza opakowanie (tzw. wrapper) dla zwykłego soecketu, który umożliwia szyfrowanie i deszyfrowanie danych wysyłanych za jego pośrednictwem. Ponadto moduł ten dostarcza dodatkowe metody, jak np. `getpeercert()`, które pozwalają na pobranie certyfikatu od drugiej storny połączenia, oraz metodę `cipher()`, która pozwala pobrać informacje o algorytmie szyfrowania wykorzytywanym w bezpiecznym połączeniu.
- **Server Name Indication (SNI) certyfikatu SSL** - jest to rozszerzeniem protokołu SSL / TLS, które pozwala SSL / TLS klienta (na przykład, przeglądarce) podać dokładną nazwę hosta próbującą połączyć się na początku procesu handshaking TLS / SSL. Po stronie serwera HTTP, pozwala na wielokrotny połączenie hosta serwera zabezpieczonych stron internetowych, które to wykorzystują ten sam adres IP i numer portu, bez konieczności używania dedykowanych numerów IP. SNI został zaprojektowany, aby umożliwić większą elastyczność i lepszą skalowalność przy użyciu protokołu SSL / TLS w środowisku multi-tenant hosting

Przykłady:

1. Połączenie klienta z serwerem za pomocą bezpiecznych gniazd (szyfrowanie, brak weryfikacji tożsamości serwera):

```
1  import socket, ssl
2  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  sock.connect((HOST, PORT))
4  context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
5  if ssl.HAS_SNI:
6         secure_sock = context.wrap_socket(sock, server_hostname=HOST)
7  else :
8         secure_sock = context.wrap_socket(sock)
9  secure_sock.write(' ... ')
10 secure_sock.read(1024)
11 secure_sock.close()
12 sock.close()
```

2. Połączenie klienta z serwerem za pomocą bezpiecznych gniazd (szyfrowanie, weryfikacja tożsamości serwera):

```
1  import socket, ssl
2  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  sock.connect((HOST, PORT))
4  context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
5  context.verify_mode = ssl.CERT_REQUIRED
6  context.load_verify_locations('./GeoTrustGlobalCA.pem')
7  if ssl.HAS_SNI:
8         secure_sock = context.wrap_socket(sock, server_hostname=HOST)
9  else:
10        secure_sock = context.wrap_socket(sock)
11 cert = secure_sock.getpeercert()
12 if not cert or ssl.match_hostname(cert, HOST):
13        raise Exception("Error" )
14 secure_sock.write(' ... ')
15 secure_sock.read(1024)
16 secure_sock.close()
17 sock.close()
```

W Pythonie istnieją 2 metody na zabezpieczenie socketu:

- Wykorzystanie funkcji `wrap_socket` oraz jej argumentów:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

secure_sock = ssl.wrap_socket(sock, server_side=False, ca_certs="server.pem",
                      certfile="client.pem", keyfile="client.key",
                      cert_reqs=ssl.CERT_REQUIRED,
                      ssl_version=ssl.PROTOCOL_TLSv1_2)
```

- Wykorzystanie klasy `SSLContext` oraz jej funkcji:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.load_verify_locations('server.pem')
context.load_cert_chain(certfile="client.pem", keyfile="client.key")

secure_sock = context.wrap_socket(sock, server_side=False)
```

Bezpieczne gniazda TLS/SSL (C/C++)

- Biblioteka OpenSSL w C/C++ umożliwia wykorzystanie w aplikacji gniazd zabezpieczonych za pomocą TLS/SSL (tzw. Secure Sockets Layer). Połączenia wykorzystujące bezpieczne gniazda umożliwiają szyfrowanie połączeń oraz autentykację zarówno strony serwera jak i klienta.
- OpenSSL to wieloplatformowa, otwarta implementacja protokołów SSL (wersji 2 i 3) i TLS (wersji 1) oraz algorytmów kryptograficznych ogólnego przeznaczenia.

Przykłady:

1. Połączenie klienta z serwerem za pomocą bezpiecznych gniazd (szyfrowanie, brak weryfikacji tożsamości serwera, po uzupełnieniu kodu kompilujemy za pomocą `g++ test.cpp -o test -lcrypto -lssl`):

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

void init_openssl()
{
    SSL_library_init();
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

void cleanup_openssl()
{
    EVP_cleanup();
}

int main(int argc, char **argv)
{
        int sock, port = 465;
        // ...
        std::string servername = "umcs.pl", greeting = "EHLO user\r\n";
        sock = socket(AF_INET, SOCK_STREAM, 0);
        connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

        init_openssl();

        SSL_CTX *ctx = SSL_CTX_new(TLSv1_2_method());
        if (!ctx) return -1;
```

```
28
29          SSL *ssl = SSL_new(ctx);
30          SSL_set_fd(ssl, sock);
31
32          SSL_ctrl(ssl, SSL_CTRL_SET_TLSEXT_HOSTNAME, TLSEXT_NAMETYPE_host_name,
33                                  (void*)servername.c_str());
34
35          int result = SSL_connect(ssl);
36          if (result == 0){
37                  long error = ERR_get_error();
38                  const char* error_str = ERR_error_string(error, NULL);
39                  printf("%s\n", error_str);
40                  return -1;
41          }
42
43          SSL_write(ssl, greeting.c_str(), greeting.size());
44          SSL_read(ssl, buff, 1024);
45
46          printf("%s\n", buff);
47
48          close(sock);
49          SSL_CTX_free(ctx);
50
51          cleanup_openssl();
52
53          return 0;
54  }
```

2. Połączenie klienta z serwerem za pomocą bezpiecznych gniazd (szyfrowanie, weryfikacja tożsamości serwera, po uzupełnieniu kodu kompilujemy za pomocą `g++ test.cpp -o test -lcrypto -lssl`):

```
1   #include <openssl/ssl.h>
2   #include <openssl/err.h>
3
4   void ShowCerts(SSL* ssl)
5   {
6       X509 *cert = SSL_get_peer_certificate(ssl);
7
8       if ( cert != NULL )
9       {
10          char* subject = new char[ 2048 + 1 ];
11          printf("\nServer certificates:\t\t%s\n",
12                          X509_NAME_oneline( X509_get_subject_name( cert ), subject, 2048 ));
13          delete subject;
14
15          char* issuer = new char[ 2048 + 1 ];
16          printf("Issuer:\t\t\t\t%s\n\n",
17                          X509_NAME_oneline( X509_get_issuer_name( cert ), issuer, 2048 ));
18          delete issuer;
19      }
20      else
21          printf("No certificates.\n");
22  }
23
24  void init_openssl()
25  {
26      SSL_library_init();
27      SSL_load_error_strings();
28      OpenSSL_add_ssl_algorithms();
29  }
30
31  void cleanup_openssl()
32  {
33      EVP_cleanup();
34  }
35
36
37
```

```
38 int main(int argc, char **argv)
39 {
40         int sock, port = 465;
41         // ...
42         std::string servername = "umcs.pl", greeting = "EHLO user\r\n";
43         sock = socket(AF_INET, SOCK_STREAM, 0);
44         connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr));
45
46         init_openssl();
47
48         SSL_CTX *ctx = SSL_CTX_new(TLSv1_2_method());
49         if (!ctx) return -1;
50
51         SSL *ssl = SSL_new(ctx);
52         SSL_set_fd(ssl, sock);
53
54         SSL_ctrl(ssl, SSL_CTRL_SET_TLSEXT_HOSTNAME, TLSEXT_NAMETYPE_host_name,
55                               (void*)servername.c_str());
56
57         int result = SSL_connect(ssl);
58         if (result == 0){
59                 long error = ERR_get_error();
60                 const char* error_str = ERR_error_string(error, NULL);
61                 printf("%s\n", error_str);
62                 return -1;
63         }
64
65         SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT,0);
66
67
68         result = SSL_CTX_load_verify_locations(ctx, "GeoTrustGlobalCA.pem", NULL);
69
70         if(!(1 == result))
71         {
72                 printf("Error in SSL_CTX_load_verify_locations\n");
73                 return -1;
74         }
75
76         X509* cert = SSL_get_peer_certificate(ssl);
77         if(NULL == cert)
78         {
79                 printf("Error in SSL_get_peer_certificate\n");
80                 return -1;
81         }
82
83         ShowCerts(ssl);
84
85         result = SSL_get_verify_result(ssl);
86         if(!(X509_V_OK == result))
87         {
88                 printf("Error in SSL_get_verify_result\n");
89                 return -1;
90         }
91
92         SSL_write(ssl, greeting.c_str(), greeting.size());
93         SSL_read(ssl, buff, 1024);
94
95         printf("%s\n", buff);
96
97         close(sock);
98         SSL_CTX_free(ctx);
99
100        cleanup_openssl();
101
102        return 0;
103 }
```

Uwaga W poniższych zadaniach zakładamy, że wszystkie połączenia powinny być połączeniami szyfrowanymi. Jeśli w zadaniu trzeba napisać serwer, powinien on obsługiwać kilku klientów jednocześnie.

1. Napisz program klienta, który połączy się z serwerem ESMTP, a następnie wyśle wiadomość e-mail używając komend protokołu ESMTP. O adres nadawcy, odbiorcy (odbiorców), temat wiadomości i jej treść zapytaj użytkownika.

   Do wykonania zadań możesz wykorzystać konta pocztowe oraz program z obsługujący protokół SMTP z laboratorium nr 6:

   - `pas2017@interia.pl` z hasłem `P4SInf2017`, serwer `interia.pl`, port `465`.
   - `pasinf2017@interia.pl` z hasłem `P4SInf2017`, serwer `interia.pl`, port `465`.
   - `pasinf2017@gmail.com` z hasłem `P4SInf2017`, serwer `smtp.gmail.com` , port `465`.

   Do ewentualnego przetestowania komend serwera SMTP na porcie `465` nie możemy wykorzystać klienta Telnet, ponieważ na porcie `465` serwer wykorzystuje gniazda zabezpieczone. Aby przetestować komendy serwera SMTP na porcie zabezpieczonym, możemy użyć klienta OpenSSL: `openssl s_client -connect interia.pl:465`.

   Napisz 2 wersje programu:

   (a) Klient nie weryfikuje tożsamości serwera

   (b) Klient weryfikuje tożsamość serwera

2. Pod adresem `httpbin.org` na porcie TCP o numerze `443` działa serwer obsługujący protokół HTTPS w wersji `1.1`. Pod odnośnikiem `/html` udostępnia prostą stronę HTML. Napisz program klienta, który połączy się z serwerem, a następnie pobierze treść strony i zapisze ją na dysku jako plik z rozszerzeniem `*.html`. Spreparuj żądanie HTTP tak, aby serwer myślał, że żądanie przyszło od przeglądarki `Safari 7.0.3`.

   Napisz 2 wersje programu:

   (a) Klient nie weryfikuje tożsamości serwera

   (b) Klient weryfikuje tożsamość serwera

3. Na serwerze `chat.freenode.net` na zabezpieczonym porcie TCP o numerze `7000` działa serwer obsługujący protokół IRC. Napisz program chat bota (klienta implementującego protokół IRC (RFC 1459)), który na wysłaną do niego wiadomość w formacie `pogoda miasto` (np. `pogoda lublin`) będzie odpowiadał informacją o aktualnej pogodzie w danym mieście.

   Do pobrania danych dotyczących pogody, możesz wykorzystać serwis `openweathermap.org`, który udostępnia API pod adresem https://openweathermap.org/current. Nie musisz rejestrować nowego konta, możesz wykorzystać następujący klucz API: `d4af3e33095b8c43f1a6815954face64`. (Klucz został wygenerowany dla adresu e-mail: `pasinf2017@gmail.com`, nazwa użytkownika to `PAS2017`, hasło `P4SInf2017`.)

   Nie wykorzystuj żadnych dodatkowych bibliotek (jedyne dozwolone to te wykorzystujące gniazda, i ewentualnie odpowiedzialne za parsowanie formatu `xml` i/lub `json`).

   Dokonaj autoryzacji serwera; zweryfkikuj jego tożsamość (tzn. sprawdź, czy certyfikat przedstawiany przez serwer został wydany (pole `issuer`) przez `Let's Encrypt Authority X3`. Zaufany certyfikat root CA (`DST Root CA X3`), możesz pobrać stąd.

4. Napisz program klienta, który połączy sie z serwerem TCP działającym pod adresem `212.182.24.27` na porcie `29443`, a następnie będzie w pętli wysyłał do niego tekst wczytany od użytkownika, i odbierał odpowiedzi. Certyfikat potrzebny do autoryzacji serwera znajduje się na stronie przedmiotu. Podejrzyj komunikację z serwerem przy użyciu Wiresharka (`tcp.dstport == 29443`).

   Napisz 2 wersje programu:

   (a) Klient nie weryfikuje tożsamości serwera

   (b) Klient weryfikuje tożsamość serwera

5. Napisz program serwera, który działajac pod adresem `127.0.0.1` oraz na określonym porcie TCP, dla podłaczającego się klienta, będzie odsyłał mu przesłaną wiadomość (tzw. serwer echa). Serwer powinien wykorzystywać samodzielnie podpisany (self-signed) certyfikat, który możesz wygenerować za pomocą biblioteki `OpenSSL`.

6. Napisz program serwera, który działajac pod adresem `127.0.0.1` oraz na określonym porcie TCP, dla podłaczającego się klienta, będzie odsyłał mu przesłaną wiadomość (tzw. serwer echa). Napisz program klienta, który połączy sie z serwerem TCP działającym pod adresem `127.0.0.1` na określonym porcie TCP, a następnie będzie w pętli wysyłał do niego tekst wczytany od użytkownika, i odbierał odpowiedzi.

   Po stronie klienta zweryfikuj tożsamość serwera. Po stronie serwera zweryfikuj tożsamość klienta. Dla serwera i klienta wygeneruj certyfikaty (self-signed) i klucze za pomocą biblioteki `OpenSSL`. Podejrzyj komunikację pomiędzy klientem a serwerem przy użyciu Wiresharka (`tcp.dstport == X`).