

# OpenSSL Programmierung

Florian Westphal

23. Juni 2005

```
1024D/1551F8FF <westphal@foo.fh-furtwangen.de>  
5E15 6DC2 67E2 FAC7 BBF3  
56B0 1003 1058 1551 F8FF
```

# Themenübersicht

- Kurzvorstellung SSL
- Die wichtigsten Funktionen und Datenstrukturen
- Ein einfacher SSL-Client
- OpenSSL Fehlerbehandlung
- PRNG
- OpenSSL und Nonblocking-IO
- Zertifikate
- BIO Layer

# Kurzvorstellung SSL

Zweck: Datenverschlüsselung zwischen zwei Hosts

Vorgehen:

- Aushandeln der zu verwendenden Chiffren
- Optional: Gegenseitige Authentifikation mit Zertifikaten
- Verschlüsselten Tunnel aufbauen

## Kurzvorstellung SSL (forts).

Die wichtigsten Protokollversionen:

- SSLv2 (Netscape 1994)
- SSLv3 (Netscape 1996)
- TLSv1 (Veröffentlichung durch IETF 1999, RFC 2246)
- TLS Extensions: 2003 (RFC 3546)

# Kurzvorstellung OpenSSL

- Implementation v. SSL (v2/v3), TLS v1
- 'General purpose' Crypto-Bibliothek
- Eigene Buffer bzw. I/O-Abstraktion: BIO
- API umfasst > 200 Funktionen
- Aber: Nicht die einzige SSL Bibliothek (GNUTLS, MatrixSSL, . . . )

## Ein einfacher TCP Client

```
int main(int argc, char *argv[]) {
    int fd, err;
    struct addrinfo *a;
    static char buf[1000];
    [...]
    err = getaddrinfo(argv[1], argv[2], NULL, &a);
    [...]
    fd = socket(a->ai_family, a->ai_socktype, a->ai_protocol);
    connect(fd, a->ai_addr, a->ai_addrlen);
    [...]
    fgets(buf, sizeof buf, stdin);
    write(fd, buf, strlen(buf));
    read(fd, buf, sizeof buf);
    [...]
```

## Die Wichtigsten API-Funktionen

`SSL_library_init()`

... muss zuerst aufgerufen werden.

`SSL_load_error_strings()`

... lädt Fehlermeldungen.

```
int SSL_write(SSL *ssl, const void *buf, int num);
```

```
int SSL_read(SSL *ssl, void *buf, int num);
```

```
int SSL_connect(SSL *ssl);
```

```
int SSL_accept(SSL *ssl);
```

... die 'grundlegenden' (IO-)Funktionen.

# Datenstrukturen

- `SSL_METHOD` – beschreibt interne SSL Funktionen, die die verschiedenen Protokolle (SSLv2, TLS, . . . ) implementieren.
- `SSL_CTX` – (SSL Context) Globale Kontextstruktur, wird von Client oder Server gewöhnlich einmal bei Programmstart erzeugt.
- `SSL` – (SSL Connection) Hauptstruktur, pro SSL-Verbindung eine Struktur; wird von `SSL_CTX` abgeleitet.
- `SSL_SESSION` und `SSL_CIPHER`– (SSL Session) TLS/SSL session-Details für eine Verbindung: `SSL_CIPHERs`, Client/Server Zertifikate, . . .

## struct SSL und struct SSL\_CTX

Die SSL-Struktur enthält die Daten (Timeout, Optionen, ..) *einer* TLS/SSL Verbindung.

Um SSL-Strukturen zu erzeugen muss zuerst ein SSL-Kontext erstellt werden:

```
struct SSL_CTX *ssl_ctx_client;  
ssl_ctx_client = SSL_CTX_new( SSLv23_client_method() );
```

In diesen 'Kontext' *können* nun z.B. Zertifikate geladen werden. `SSL_CTX_free()` muss verwendet werden, um den SSL Kontext freizugeben.

## Ein einfacher SSL Client

Mittels des SSL\_CTX kann nun struct SSL erzeugt werden:

```
struct SSL *ssl;  
ssl = SSL_new(ssl_ctx_client);
```

Nun muss noch der Deskriptor gesetzt werden:

```
SSL_set_fd(ssl, fd);
```

Anschließend kann die Struktur in Funktionen wie SSL\_write(), SSL\_read(), SSL\_connect(), etc. verwendet werden. Die Struktur kann mit SSL\_free() freigegeben werden.

## Wdh: Ein einfacher TCP Client

```
int main(int argc, char *argv[]) {
    int fd, err;
    struct addrinfo *a;
    static char buf[1000];
    [...]
    err = getaddrinfo(argv[1], argv[2], NULL, &a);
    [...]
    fd = socket(a->ai_family, a->ai_socktype, a->ai_protocol);
    connect(fd, a->ai_addr, a->ai_addrlen);
    [...]
    write(fd, buf, strlen(buf));
    read(fd, buf, sizeof buf);
    [...]
```

## Ein einfacher SSL Client

```
#include <ssl/openssl.h>
int main(int argc, char *argv[]) {
/* int fd, err; ... alles wie gehabt */
  SSL_CTX * ssl_ctx_client;
  SSL *ssl;
  [...]
/* getaddrinfo(), socket().. */
  [...]
  connect(fd, a->ai_addr, a->ai_addrlen));
```

SSL-Initialisierung:

```
SSL_library_init();
SSL_load_error_strings();
[...]
```

## Ein einfacher SSL Client (forts.)

Erzeugen des SSL-Kontextes, Client Mode. . .

```
ssl_ctx_client = SSL_CTX_new( SSLv23_client_method() );  
if (!ssl_ctx_client) return 111;
```

. . . und der SSL-Struktur.

```
ssl = SSL_new(ssl_ctx_client);  
if (!ssl) return 111;
```

Zuletzt setzen wir den Deskriptor:

```
if (SSL_set_fd(ssl, fd) != 1) return 111;
```

## Ein einfacher SSL Client (forts.)

Nun wird der SSL-Handshake initiiert:

```
if (1 != SSL_connect(ssl)) return 111;
```

Anschließend kann mit `SSL_read()` bzw. `SSL_write()` gelesen/geschrieben werden.

```
fgets(buf, sizeof buf, stdin);  
SSL_write(ssl, buf, strlen(buf));
```

```
b_read = SSL_read(ssl, buf, sizeof buf);  
if (b_read > 0)  
    write(1, buf, b_read);  
return 0;  
}
```

## Ein einfacher SSL Server

Ein-Server funktioniert *fast* genauso:  
Anstelle der Client-Methoden werden die entsprechenden Server-Methoden verwendet.

```
SSL_CTX *ssl_ctx_server = SSL_CTX_new(SSLv23_server_method());  
[..]  
fd = accept(sock, &sa, &sa_len);  
[..]  
ssl = SSL_new(ssl_ctx_server);  
SSL_set_fd(ssl, fd);  
[..]  
SSL_accept(ssl);
```

# Rückgabewerte der (wichtigsten) SSL I/O Funktionen

- `SSL_connect()`, `SSL_accept()`, `SSL_do_handshake()`:
  - return 1: OK
  - return 0: SSL/TLS shutdown.
  - return  $< 0$ : (Fatal) Fehler.
- `SSL_read()`, `SSL_peek()`, `SSL_write()`:
  - return  $> 0$ : Anzahl übertragener Bytes
  - return 0: SSL/TLS shutdown.
  - return  $< 0$ : (Fatal) Fehler.

Wie aber kann man eine 'richtige' Fehlermeldung erzeugen?

## OpenSSL Fehlerbehandlung

Schlägt ein Aufruf der OpenSSL Bibliothek fehl, wird das üblicherweise über den Rückgabewert signalisiert. Zusätzlich wird ein Fehlercode in einer Warteschlange (*per Thread*) abgelegt.

```
#include <openssl/err.h>
unsigned long ERR_get_error(void);
```

Gibt den Fehlercode zurück oder 0 falls die Thread Error Queue leer ist.

## OpenSSL Fehlerbehandlung: Error Queue

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

Liefern eine 'Human Readable' Meldung in buf, Format:

```
error:[error code]:[library name]:[function name]:[reason string]
```

Es gibt auch Funktionen um z.B. nur den Funktionsnamen zu erhalten.  
Siehe dazu `ERR_error_string(3)`.

Ein Wrapper in einer Applikation könnte z.B. so aussehen:

```
char * myssl_geterrstr(void) {  
    unsigned long err = ERR_get_error();  
    return err ? ERR_error_string(err, NULL) : NULL;  
}
```

## OpenSSL Fehlerbehandlung: Error Queue (forts.)

Dummerweise können manche Aufrufe fehlschlagen *ohne* das anschließend etwas in der Queue steht.

Ein weiterer Nachteil: Es gibt keinen eleganten Weg, von `ERR_get_error()` gelieferte Fehlercodes in der Applikation auszuwerten.

**Wie kann man nach fehlgeschlagenen SSL-Funktionen wie z.B `SSL_write()` die Ursache ermitteln?**

## OpenSSL Fehlerbehandlung: `SSL_get_error`

`SSL_get_error()` liefert einen Ergebnis-Code nach einem vorangehenden Aufruf von `SSL_write`, `SSL_connect`, ...

```
int SSL_get_error(SSL *ssl, int ret);
```

Ausser `ssl` und `ret` wird auch die Error Queue untersucht. Im Klartext:  
**Wenn die Error Queue nicht leer ist funktioniert es nicht!**

Der Rückgabewert enthält genauere Informationen über den aufgetretenen Fehler. Der Wert `SSL_ERROR_NONE` wird geliefert, wenn überhaupt kein Fehler aufgetreten ist.

Der Rückgabewerte von `SSL_get_error` kann z.B. in einem switch statement verarbeitet werden.

## Die wichtigsten `SSL_get_error` Rückgabewerte

`SSL_ERROR_ZERO_RETURN`

TLS/SSL-Verbindung wurde geschlossen. Abhängig von der verwendeten SSL-Version ist der darunterliegende Deskriptor noch geöffnet.

`SSL_ERROR_WANT_READ`, `SSL_ERROR_WANT_WRITE`

Die Operation konnte nicht abgeschlossen werden, später nochmal versuchen

`SSL_ERROR_WANT_X509_LOOKUP`

`client_cert_cb()` will erneut aufgerufen werden.

`SSL_ERROR_SSL`

Fehler in der SSL-Bibliothek. (z.B. Protokollfehler)

## Die wichtigsten `SSL_get_error` Rückgabewerte (forts.)

`SSL_ERROR_SYSCALL`

I/O Fehler. Falls die Error-Queue leer ist (d.h. `ERR_get_error()==0`) dann gilt: War der Rückgabewerte der Ursprünglichen Funktion (= letzter Parameter von `SSL_get_error()` . . .

- 0: EOF
- -1: low-level Fehler in einem Syscall. Jetzt darf man `errno` auswerten.

Der Vollständigkeit halber:

`SSL_ERROR_WANT_CONNECT`, `SSL_ERROR_WANT_ACCEPT`

Tritt nur bei zugrundeliegenden `BIO_s_accept/BIO_s_connect()` auf.

## Beispiel: Anwendung von SSL\_get\_error

```
unsigned long sslerr;
switch (SSL_get_error( ssl, retcode )) {
  case SSL_ERROR_SYSCALL:
    if ((sslerr=ERR_get_error()))
      fprintf(stderr, "%s",ERR_error_string(sslerr,NULL));
    else switch (retcode) {
      case 0:          /* EOF */
        fputs("Client Disconnected", stderr); break;
      case -1:
        fprintf(stderr,"write: %s", strerror(errno));
    }
    SSL_shutdown(ssl); break;
  case SSL_ERROR_SSL:
    SSL_shutdown(ssl);
  [..]
```

# PRNG

Auf 'modernen' Plattformen verwendet die OpenSSL-Bibliothek `/dev/{u,a,}random` um den internen Entropiepool zu füllen. Auf andern Plattformen muss dies durch die Applikation selbst geschehen.

```
#include <openssl/rand.h>
int RAND_status(void);
```

Gibt 1 zurück, wenn der PRNG ausreichend mit Daten versorgt ist.

Funktionen, mit denen eine Applikation den Pool füllen kann (Auswahl):

```
void RAND_seed(const void *buf, int num);
void RAND_add(const void *buf, int num, double entropy);

int RAND_egd(const char *path);
```

# Non-Blocking I/O

Der Aufruf

```
read(fd, buf, sizeof buf);
```

blockiert 'normalerweise' so lange, bis tatsächlich Daten vorliegen. Manchmal ist dieses Verhalten jedoch unerwünscht. Bei Named Pipes kann dieses Verhalten mit dem `O_NONBLOCK` Flag bereits beim Ausführen des `open()` abgestellt werden. Sonst: `fcntl()` (`F_SETFL` in Verbindung mit `O_NONBLOCK`).

Anschließend kehren Funktionen wie `read()` oder `write()` sofort zurück, auch wenn keine Daten gelesen bzw. geschrieben werden konnten. Falls eine solche Funktion blockiert hätte, gibt sie `-1` zurück. `errno` ist auf `EAGAIN` gesetzt.

## OpenSSL und Non-Blocking I/O

Dieses Verhalten gilt nun analog für OpenSSL Funktionen wie z.B. `SSL_read()`. Hierbei gibt die SSL I/O Funktion `-1` zurück. Ein anschließender Aufruf von `SSL_get_error` liefert entweder `SSL_ERROR_WANT_READ` oder `SSL_ERROR_WANT_WRITE`. Das bedeutet, dass eine Applikation, welche Daten lesen wollte nun gezwungen ist auf Schreibbarkeit des Socket zu testen (`select()`, `poll()`, ...) (vv).

# Zertifikate

Zertifikate werden normalerweise in der SSL\_CTX Struktur gesetzt.

```
int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx,  
                                     const char *file);  
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx,  
                                const char *file, int type);
```

Die Funktion `SSL_CTX_check_private_key(SSL_CTX*ctx)` überprüft ob Zertifikat und Key zueinander passen.

Es existieren auch entsprechende auf SSL\* Strukturen operierende Funktionen. (Ohne `_CTX` im Namen).

## Zertifikatskontrolle

```
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode,  
                        int (*verify_callback)(int, X509_STORE_CTX *));
```

Das Verhalten der Funktion wird via mode kontrolliert.

SSL\_VERIFY\_NONE

Server: Keine Zertifikate

Client: Server schickt Zertifikat, Überprüfung wird durchgeführt

SSL\_VERIFY\_PEER

Server: Schickt Zertifikatanfrage an Peer, Überprüfung wird durchgeführt

Client: Serverzertifikat wird überprüft

## Zertifikatskontrolle (forts.)

SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT

Falls Peer kein Zertifikat vorweist → disconnect (Nur Server).

SSL\_VERIFY\_CLIENT\_ONCE

Nur einmalige Überprüfung des Zertifikates (Nur Server).

**Aber woher weiß OpenSSL, welche Zertifikate gültig sind?**

## Zertifikatskontrolle (forts.)

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx,  
    const char *CAfile, const char *CApath);
```

gibt an, wo die vertrauenswürdigen CA-Zertifikate gespeichert sind. Die Funktion `X509_load_crl_file` lädt eine Liste von widerrufenen Zertifikaten, d.h. von Zertifikaten die zwar von einer der „Vertrauenswürdigen“ CAs signiert sind, aber trotzdem nicht akzeptiert werden sollen.

## Zertifikatskontrolle: `verify_callback`

Zur Erinnerung:

```
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode,  
    int (*verify_callback)(int, X509_STORE_CTX *));
```

`verify_callback` kontrolliert das Verhalten falls `SSL_VERIFY_PEER` gesetzt ist. Falls ein Checking-Schritt fehlschlägt, wird `verify_callback` mit `preverify_ok=0` erneut aufgerufen (Jetzt kann man z.B. den Fehler lokalisieren).

## Beispiel: Zertifikatlisten laden

```
X509_STORE *store;
X509_LOOKUP *lookup;

SSL_CTX_load_verify_locations(ctx, CAfilename, NULL);
store = SSL_CTX_get_cert_store(ctx);
lookup = X509_STORE_add_lookup(store, X509_LOOKUP_file());
if (!lookup) return -1;
if (1 != X509_load_crl_file(lookup, CRLfilename,
    X509_FILETYPE_PEM)) return -1;
X509_STORE_set_flags(store,
    X509_V_FLAG_CRL_CHECK | X509_V_FLAG_CRL_CHECK_ALL);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, ssl_verify_cb);
return 0;
}
```

## Beispiel: Verify Callback

```
int verify_callback(int preverify_ok, X509_STORE_CTX *ctx)
{
    char buf[256];
    X509 *err_cert;
    int err, depth;

    err_cert = X509_STORE_CTX_get_current_cert(ctx);
    err = X509_STORE_CTX_get_error(ctx);
    depth = X509_STORE_CTX_get_error_depth(ctx);

    X509_NAME_oneline(X509_get_subject_name(err_cert), buf, 256);
```

## Beispiel: Verify Callback (forts.)

```
if (!preverify_ok) {
    printf("verify error:num=%d:%s:depth=%d:%s\n",
        err, X509_verify_cert_error_string(err), depth, buf);

    if (err == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT) {
        X509_NAME_oneline(X509_get_issuer_name(ctx->current_cert),
            buf, 256);

        printf("issuer= %s\n", buf);
    }
}
return Conf->Ignore_x509errors ? 1 : preverify_ok;
}
```

## Zertifikate: Nachkontrolle

Falls nicht automatisch abgebrochen wurde, kann das Ergebnis der Überprüfung mit

```
long SSL_get_verify_result(SSL *ssl);
```

ermittelt werden. Fall Ergebnis !=X509\_V\_OK: Zertifikatsprüfung negativ.

**Achtung:** X509\_V\_OK wird auch dann zurückgegeben wenn kein Zertifikat geliefert wurde!

```
X509 *SSL_get_peer_certificate(SSL *ssl);
```

...liefert NULL, falls kein Zertifikat vorhanden ist.

# BIOs

Filter/Verkettungskonzept: Es gibt 2 Typen: Source/Sink und Filter BIOs.

`BIO_new(BIO_METHOD*)` erstellt neuen BIO. `BIO_METHOD` ist dabei der gewünschte BIO-Typ. Namenskonvention:

- `BIO_METHOD* BIO_s_*()`: Source/Sink BIO
- `BIO_METHOD* BIO_f_*()`: Filter BIO

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

Zusammenfügen von BIOs bzw. Entfernen eines BIOs aus der Kette.

## Beispiel: Ein Base64 Encoder

```
#include <openssl/bio.h>
#include <openssl/evp.h>
int main(void) {
    int inlen;
    char inbuf[512];
    BIO *b64 = BIO_new(BIO_f_base64());
    BIO *in = BIO_new_fd(0, BIO_NOCLOSE);
    BIO *out = BIO_new_fd(1, BIO_NOCLOSE);
    out = BIO_push(b64, out);
    while((inlen = BIO_read(in, inbuf, sizeof inbuf)) > 0)
        BIO_write(out, inbuf, inlen);

    BIO_free_all(in);
    return 0;
}
```

## Beispiel: 3DES-CBC Ver/Entschlüsselung

```
#include <openssl/bio.h>
#include <openssl/evp.h>
int main(int argc, char *argv[]) {
    int inlen;
    char inbuf[512];
    BIO *crypt = BIO_new(BIO_f_cipher());
    BIO *in = BIO_new_fd(0, BIO_NOCLOSE);
    BIO *out = BIO_new_fd(1, BIO_NOCLOSE);
    BIO_set_cipher(crypt, EVP_des_ede3_cbc(), "key", "iv", (argc>1));
    out = BIO_push(crypt, out);
    while((inlen = BIO_read(in, inbuf, sizeof inbuf)) > 0)
        BIO_write(out, inbuf, inlen);
    BIO_free_all(in);
    return 0;
}
```

## Beispiel: 3DES-CBC Ver/Entschlüsselung (forts.)

Etwas genauer betrachtet:

```
BIO *crypt = BIO_new(BIO_f_cipher());
```

Erstellt einen BIO cipher filter

```
BIO_set_cipher(crypt, EVP_des_ede3_cbc(), "key", "iv", (argc>1));
```

Setzt den Verschlüsselungsalgorithmus sowie key & IV. Das letzte Argument ist 1 wenn verschlüsselt werden soll, sonst 0.

# Lektüre

- Die Man Pages: `ssl(3)`, `crypto(3)`, `bio(3)`, `rand(3)`, `verify(1)`, `x509(1)`.
- Beispielprogramme und Tests in den OpenSSL Quellen
- Quellen v. Programmen wie z.B. `stunnel`, `sslwrap`, . . .
- <http://www.securityfocus.com/infocus/1818> (Apache2 with SSL, bietet aber auch einen guten Überblick über die Funktionsweise von SSL)