# Utilisation de l'implémentation de DTLS fournie par OpenSSL

Using OpenSSL's DTLS Implementation

Nicolas Bernard, Université du Luxembourg

Septembre 2006

**Résumé :** Datagram Transport Layer Security (DTLS) est un protocole récent qui fournit une couche cryptographique au-dessus d'UDP. Ce protocole est implémenté dans la bibliothèque OpenSSL mais son usage n'est pas documenté. Le présent document vise à montrer à l'aide d'exemples comment utiliser cette implémentation.
**Mots-clefs :** DTLS, UDP, OpenSSL, sécurité réseau.

**Abstract :** Datagram Transport Layer Security (DTLS) is a recent prococol. It provides a cryptographical layer on the top of UDP. This protocol is implemented in the OpenSSL library but its use is not documented. This document aims to show how to use this implementation, by providing examples.
**Keywords :** DTLS, UDP, OpenSSL, network security.

SSL et TLS sont situés au-dessus de TCP et ne peuvent donc être utilisés par des applications basées sur UDP. Jusque récemment, celles-ci avaient le choix entre l'utilisation d'IPsec, souvent difficile à mettre en oeuvre par l'application elle-même et nécessitant un support du système, ou un mécanisme de chiffrement propre, avec les risques que cela implique, le développement d'un protocole de chiffrement sûr étant une tâche difficile.

Cette situation a changé avec l'arrivée de DTLS (Datagram Transport Layer Security) et la parution de la RFC 4347 qui le décrit. DTLS est une adaptation de TLS destinée à fonctionner au-dessus d'UDP.

La bibliothèque OpenSSL, qui permettait déjà l'utilisation de SSL et TLS, intègre une implémentation de DTLS depuis la version 0.9.8a, malheureusement non-documentée si ce n'est par son code source. Le but de ce document est de fournir quelques exemples pour introduire l'utilisation de DTLS avec cette implémentation et compenser ainsi quelque peu l'absence de documentation.

Note préliminaire : tous les exemples donnés ici sont écrits en C99. Si vous utilisez un compilateur plus ancien, il vous faudra probablement les modifier légèrement pour pouvoir les compiler. (Pour notre part, nous les avons compilés avec `gcc` version 3.3.5 et les options `-std=c99 -Wall -pedantic`.)

# 1 Un cas simple

Dans le cas le plus simple, celui où une application communique avec un seul correspondant à la fois via UDP (par exemple un client de téléphonie IP), l'utilisation de DTLS pour protéger les échanges est simple, et ne présente que peu de différences par rapport à l'utilisation de TLS sur une socket TCP.

Les sous-sections suivantes présentent les deux côtés de ce cas.

## 1.1 Accepter une connexion DTLS

Examinons le code du serveur. Serveur est d'ailleurs un bien grand mot, puisque le programme d'exemple présenté ici n'accepte qu'une unique connexion avant d'achever son exécution. Il est serveur uniquement en ceci qu'il attend une connexion au lieu de l'initier.

Listing 1 – « serveur » DTLS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

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

#define PORT 2053

#define SERVER 1
static int
makesock(int port)
{
        /* voir en annexe */
}

int
main()
{
        SSL_library_init ();
        SSL_load_error_strings ();

        FILE* paramfile = fopen("dh_param_1024.pem", "r");
        if (paramfile == NULL) {
                fprintf (stderr, "Error opening the DH file: %s\n",
                        strerror (errno));
                return EXIT_FAILURE;
        }

        DH* dh_1024 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
        fclose (paramfile);
```

```
        if (dh_1024 == NULL) {
39              fprintf(stderr, "Error reading the DH file\n");
                return EXIT_FAILURE;
        }
42
        int sock = makesock(PORT);

45      BIO* conn = BIO_new_dgram(sock, BIO_NOCLOSE);
        if (conn == NULL) {
                fprintf(stderr,"error creating bio\n");
48              return EXIT_FAILURE;
        }

51      SSL_CTX *ctx = SSL_CTX_new(DTLSv1_server_method());
        if (ctx == NULL) {
                ERR_print_errors_fp(stderr);
54              return EXIT_FAILURE;
        }
        SSL_CTX_set_read_ahead(ctx, 1);
57      SSL_CTX_set_cipher_list(ctx, "HIGH:MEDIUM:aNULL");
        SSL_CTX_set_tmp_dh(ctx, dh_1024);

60      SSL *ssl = SSL_new(ctx);
        if (ssl == NULL) {
                return EXIT_FAILURE;
63      }

        SSL_set_bio(ssl, conn, conn);
66      SSL_set_accept_state(ssl);

        char buf[200] = {0};
69
        int err = SSL_read(ssl, &buf, 199);
        if (err <= 0) {
72              err = SSL_get_error(ssl, err);
                fprintf(stderr, "SSL_read: error %d\n", err);
                ERR_print_errors_fp(stderr);
75              return EXIT_FAILURE;
        }
        fprintf(stderr, "%s", buf);
78
        char *c = "Hello yourself!\n";
        err = SSL_write(ssl, c, strlen(c));
81      if (err <= 0) {
                err = SSL_get_error(ssl, err);
                fprintf(stderr, "SSL_write: error %d\n", err);
84              ERR_print_errors_fp(stderr);
                if (err == SSL_ERROR_SYSCALL)
                        fprintf(stderr, "errno: %s\n", strerror(errno));
87      }

        SSL_free(ssl); /* free conn too */
90      SSL_CTX_free(ctx);
```

```
        DH_free(dh_1024);
        close(sock);
        return 0;
}
```

La première partie, entre les lignes 26 et 41, initialise OpenSSL et charge les paramètres à utiliser pour le protocole de Diffie-Hellman (un fichier tel qu'attendu ici peut être généré avec la ligne de commande `openssl dhparam -out dh_param_1024.pem -2 1024`).

À la ligne 43, on appelle la fonction `makesock` dont le code est donné en annexe (listing 7), qui crée la socket UDP et la lie à un port local. Cette fonction renvoie le descripteur correspondant à cette socket.

La partie qui nous intéresse réellement commence alors : entre la ligne 45 et la ligne 49 on crée un BIO que l'on associe à notre socket (un BIO est une couche d'abstraction entre les mécanismes d'entrées / sorties du système et l'application). On utilise la méthode `BIO_new_dgram` qui correspond à une socket UDP.

Juste après, on crée un contexte (objet `SSL_CTX`). Bien qu'écrite en C, la bibliothèque OpenSSL est relativement orientée objet. Les objets `SSL_CTX` sont des structures qui contiennent entre autres un certain nombre de pointeurs vers des fonctions de bas niveau, et permettent aux fonctions de haut niveau comme `SSL_read` et `SSL_write` de faire abstraction des détails relatifs à une version particulière de SSL ou de TLS.

On crée donc ce contexte en indiquant le rôle qu'il devra tenir – celui d'un serveur DTLS. À la ligne 57 on indique que l'on veut utiliser les chiffres symétriques forts (`HIGH`) et moyens (`MEDIUM`), ainsi que les algorithmes d'échange de clefs qui n'authentifient pas les machines (`aNULL`), c'est-à-dire le protocole de Diffie-Hellman de base (nous choisissons ce dernier car cela nous permet de nous dispenser de certificats, l'utilisation de ceux-ci étant similaire au cas classique de TLS/SSL). L'utilisation de la macro `SSL_CTX_set_read_ahead` (ligne 56) est destinée à prévenir la perte de la fin des paquets UDP.

Une fois cela fait, on peut préparer le moteur SSL pour qu'il accepte une connexion sur cette socket (lignes 60-66). On crée donc un objet `SSL` utilisant le contexte précédent (ligne 60), que l'on associe à notre BIO (ligne 65) et l'on indique explicitement que l'on joue le rôle du serveur (ligne 66).

Notre lien DTLS est maintenant prêt à être utilisé, c'est ce que l'on fait de la ligne 68 à la ligne 87. `SSL_read` est bloquant jusqu'à ce qu'il y ait des données à lire (cette fonction fera implicitement la négociation des chiffres à utiliser).

Finalement, on désalloue les structures allouées et ferme la socket (ligne 89-92).

## 1.2 Initier une connexion DTLS

Comme le montre le listing suivant, le code du client est très similaire à celui du serveur.

Listing 2 – « client » DTLS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
```

```
9  #include <unistd.h>

   #include <openssl/ssl.h>
12 #include <openssl/bio.h>
   #include <openssl/err.h>

15 #define PORT 2053

   static int
18 makesock(int port)
   {
           /* voir en annexe */
21 }

   int
24 main()
   {
           SSL_library_init ();
27         SSL_load_error_strings ();

           int sock = makesock(PORT);
30
           BIO* conn = BIO_new_dgram(sock, BIO_NOCLOSE);
           if (conn == NULL) {
33                 fprintf (stderr,"error creating bio\n");
                   return 1;
           }
36
           struct sockaddr_in dst;
           struct sockaddr* d = (struct sockaddr*) &dst;
39 #ifndef __GLIBC__
           dst.sin_len = sizeof(struct sockaddr_in);
   #endif
42         dst.sin_family = AF_INET;
           dst.sin_port = htons(PORT);
           dst.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
45
           int err = BIO_dgram_set_peer(conn, d);
           fprintf (stderr, "BIO_dgram_set_peer: %d\n", err);
48
           SSL_CTX *ctx = SSL_CTX_new(DTLSv1_client_method());
           if (ctx == NULL) {
51                 ERR_print_errors_fp(stderr);
                   return 2;
           }
54         SSL_CTX_set_read_ahead(ctx, 1);
           SSL_CTX_set_cipher_list(ctx, "HIGH:MEDIUM:aNULL");

57         SSL *ssl = SSL_new(ctx);
           if (ssl == NULL) {
                   return 3;
60         }
```

```
        SSL_set_bio(ssl, conn, conn);
63      SSL_set_connect_state(ssl);

        char *c = "Hello, enciphered World!\n";
66      err = SSL_write(ssl, c, strlen(c));
        if (err <= 0) {
                err = SSL_get_error(ssl, err);
69              fprintf(stderr, "SSL_write: error %d\n", err);
                ERR_print_errors_fp(stderr);
                if (err == SSL_ERROR_SYSCALL)
72                      fprintf(stderr, "errno: %s\n", strerror(errno));
        }

75      char buf[256] = {0};
        err = SSL_read(ssl, buf, 256);
        if (err <= 0) {
78              err = SSL_get_error(ssl, err);
                fprintf(stderr, "SSL_read: error %d\n", err);
                ERR_print_errors_fp(stderr);
81              if (err == SSL_ERROR_SYSCALL)
                        fprintf(stderr, "errno: %s\n", strerror(errno));
        }
84      fprintf(stderr, "%s\n", buf);

        SSL_free(ssl); /* free conn too */
87      SSL_CTX_free(ctx);
        close(sock);
        return 0;
90  }
```

Examinons donc uniquement les différences entre ce code et le précédent :

- on ne charge pas les paramètres pour Diffie-Helleman ;
- entre les lignes 37 et 44, on initialise une structure `sockaddr` avec l'adresse et le port du serveur auquel on veut se connecter. Bien sûr, dans un client réel, ce serait fait d'une manière plus habituelle avec, par exemple, `getaddrinfo`, en fonction des paramètres de la ligne de commande. Ici nous nous contentons d'utiliser l'adresse de boucle locale et le port défini par la macro `PORT` ;
- cette structure est passée en second paramètre de la macro `int BIO_dgram_set_peer(BIO* b, struct sockaddr* addr)` ligne 46, qui « intègre » cette adresse comme étant la destination du bio `conn`. En effet, souvenez-vous que la fonction `SSL_write`, contrairement à `sendto` ne permet pas de spécifier l'adresse de destination.
- à la ligne 63, `SSL_set_connect_state` remplace le `SSL_set_accept_state` du serveur ;
- l'ordre des `SSL_read` et `SSL_write` est inversé, puisqu'ils doivent correspondre à ceux du serveur.

## 1.3 Différences entre l'utilisation de DTLS et TLS

Dans cet exemple, il n'y a pas de différence majeure avec que que l'on aurait fait si l'on avait voulu à la place utiliser TLS sur TCP. Pour le client, il n'y a guère que le type de la socket et le paramètre passé à `SSL_CTX_new` qui changent, tandis que le serveur se trouve plutôt simplifié puisque la partie de mise en place de la socket se passe des appels à `listen`(2)

et que l'on n'utilise pas `SSL_accept`.

Cependant, les sockets UDP sont souvent utilisées d'une manière différentes des sockets TCP, ce que ne prend pas en compte l'exemple précédent. C'est de ce cas que traite la section suivante.

## 2 Multiplexage sur une socket

Malheureusement le cas simple présenté précédemment est loin d'être représentatif de ce qui est utilisable avec la plupart des applications utilisant UDP.

En effet, une application UDP utilisera souvent une unique socket pour communiquer avec un grand nombre de clients (alors qu'une application TCP doit utiliser une socket par connexion).

Une telle utilisation est difficilement compatible avec l'interface de base proposée par OpenSSL, celle-ci étant intrinsèquement orientée connexion, bien qu'au-dessus d'UDP. Il faut dire que DTLS même est orienté connexion, puisque qu'il est nécessaire que les deux parties qui communiquent conviennent d'une clef de session. Cependant, pour la plupart des applications UDP qui fonctionnent selon ce modèle, il serait difficile de passer à un modèle avec une socket par client, car elles ont généralement plus de clients qu'un programme ne peut ouvrir de sockets sur la plupart des systèmes.

L'architecture d'OpenSSL est cependant construite de manière à ce que l'utilisateur puisse gérer les détails des communications s'il le désire. Dans notre cas, cela nous permet de multiplexer plusieurs connexions DTLS sur une seule socket UDP, voire même de mélanger des communications sécurisées avec DTLS avec des communications non sécurisées.

En effet, alors que l'utilisation standard d'une socket TLS avec OpenSSL peut ressembler à ce que l'on voit figure 1, il est possible de prendre le contrôle des écritures de bas niveau pour aboutir au schéma représenté figure 2.

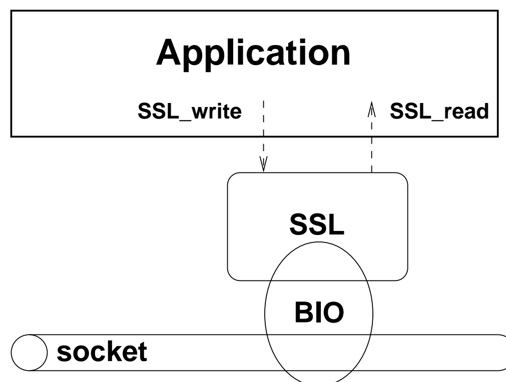


Figure 1 – Utilisation classique d'OpenSSL. La socket est utilisée implicitement.

Nous pouvons donc écrire un module `dtlsplex.c` définissant les fonctions données dans le fichier d'en-tête `dtlsplex.h` (listing 3). On le voit, ce module fournit deux fonctions, `dtlsrecvfrom` et `dtlssendto`, l'idée étant que ces fonctions puissent être utilisées comme les `recvfrom` et `sendto` classiques. Bien que ce soit souvent le cas, **un certain nombre d'hypothèses sur les fonctions classiques ne sont plus valides**, comme nous le verrons ci-dessous, la principale différence étant que `dtlssendto` échoue sur le client la première

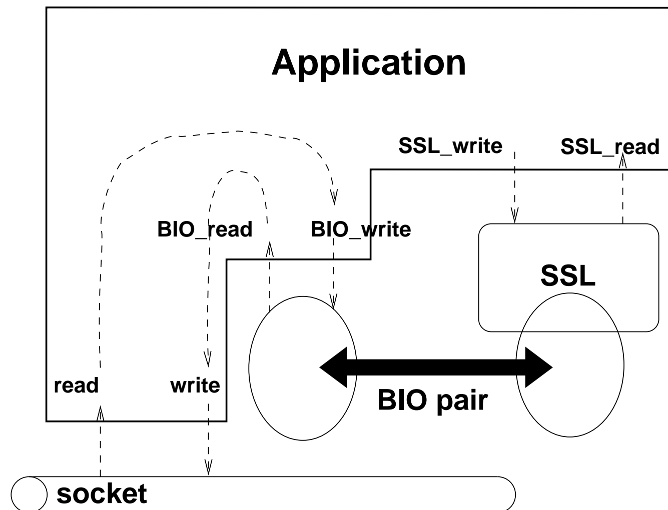


FIGURE 2 – Utilisation d'un `BIO_pair` pour utiliser SSL comme un filtre et gérer explicitement la socket.

fois qu'elle est utilisée (elle retourne alors 0), il faut alors la rappeler **après avoir appelé** `dtlsrecvfrom` afin que la session DTLS soit établie.

Listing 3 – `dtlsplex.h`

```
#ifndef _DTLSPLEX_H_
#define _DTLSPLEX_H_

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

#define MTU 1500
ssize_t dtlsrecvfrom(int s, void *buf, size_t len, int flags,
                     struct sockaddr *from, socklen_t *fromlen);
ssize_t dtlssendto(int s, const void *msg, size_t len, int flags,
                   const struct sockaddr *to, socklen_t tolen);

struct dtls_peer {
        struct sockaddr_storage addr;
        BIO* bio;
        BIO* _b2;
        SSL* ssl;
};

#endif /* ! _DTLSPLEX_H_ */
```

Avec ces fonctions, les codes des fonctions `main` de notre client et de notre serveur deviennent respectivement ceux des listings 4 et 5. Notez que le serveur peut cette fois traiter un nombre quelconque de connexions.

Listing 4 – La fonction `main` du client en utilisant `dtlsrecvto` et `dtlssendto`

```
int
main()
```

```
{
        SSL_library_init ();
        SSL_load_error_strings ();

        int sock = makesock();

        struct sockaddr_in dst;
        struct sockaddr* d = (struct sockaddr*) &dst;
#ifndef __GLIBC__
        dst.sin_len = sizeof(struct sockaddr_in);
#endif
        dst.sin_family = AF_INET;
        dst.sin_port = htons(PORT);
        dst.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

        int ret;

        char buf[256] = {0};
        struct sockaddr_storage s;
        socklen_t slen = 0;

        do {
                char ch[] = "Hello, enciphered World!\n";
                int l = strlen(ch);
                ret = dtlssendto(sock, ch, l, 0, d, sizeof dst);
                if (ret < 0)
                        fprintf(stderr, "dtlssendto returned %d\n", ret);
                int ret2 = dtlsrecvfrom(sock, buf, 256, 0,
                                        (struct sockaddr*) &s, &slen);
                fprintf(stderr, "dtlsrecvfrom returned %d\n", ret2);
                if (ret2 > 0)
                        fprintf(stderr, "%s\n", buf);
        } while (ret == 0);

        int ret2 = dtlsrecvfrom(sock, buf, 256, 0,
                                (struct sockaddr*) &s, &slen);
        fprintf(stderr, "dtlsrecvfrom returned %d\n", ret2);
        if (ret2 > 0)
                fprintf(stderr, "%s\n", buf);

        return 0;
}
```

Listing 5 – La fonction `main` du serveur en utilisant `dtlsrecvto` et `dtlssendto`

```
int
main()
{
        SSL_library_init ();
        SSL_load_error_strings ();

        int sock = makesock();

        while(1) {
```

```c
        char buf[256] = {0};
        struct sockaddr_storage s;
        socklen_t slen = 0;
        int ret = dtlsrecvfrom(sock, buf, 256, 0,
                               (struct sockaddr*) &s, &slen);
        fprintf(stderr, "dtlsrecvfrom returned %d\n", ret);
        fprintf(stderr, "%s", buf);

        char ch[] = "Hello Yourself";
        ret = dtlssendto(sock, ch, strlen(ch), 0,
                         (struct sockaddr*) &s, slen);
        fprintf(stderr, "dtlssendto returned %d\n", ret);
    }

    return 0;
}
```

Examinons maintenant comment sont implémentées ces fameuses fonctions `dtlssendto` et `dtlsrecv`. Tout d'abord, on note que comme DTLS est orienté connexion, il faut pouvoir garder en mémoire l'état d'une connexion. La structure `dtls_peer`, définie dans le fichier d'en-tête contient les données de cet état. Notre module suppose qu'il existe deux fonctions selon les prototypes donnés dans le fichier `peer.h` suivant :

```c
#ifndef _PEER_H_
#define _PEER_H_

/** store "somewhere" the peer at IP address addr */
struct dtls_peer* addpeer(struct dtls_peer* p, const struct sockaddr* addr);

/** returns the data associated to the peer of address addr,
    which must have been previously stored using addpeer (else,
    it returns NULL */
struct dtls_peer* getpeer(const struct sockaddr* addr, int addrlen);

#endif
```

Ces fonctions devront donc être définies. Étant donné que le stockage variera en fonction des besoins de l'application, le code de ces fonctions n'est pas fourni ici.

Le listing suivant nous montre donc comment opèrent ces fonctions :

Listing 6 – Le module `dtlsplex.c`

```c
#include <sys/types.h>
#include <sys/socket.h>

#include <assert.h>
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

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

```
#include <openssl/dh.h>
#include <openssl/err.h>
15
#include "dtlsplex.h"
#include "netmisc.h"
18 #include "peer.h"


21


static SSL*
24 SSL_create(SSL_CTX* ctx, BIO* conn, bool client)
{
        assert(ctx != NULL);
27      assert(conn != NULL);

        SSL *ssl = NULL;
30      ssl = SSL_new(ctx);
        if (ssl == NULL) {
                return NULL;
33      }
        if (client)
                SSL_set_connect_state(ssl);
36      else
                SSL_set_accept_state(ssl);

39      SSL_set_bio(ssl, conn, conn);

        return ssl;
42 }

static SSL_CTX*
45 createctx(void)
{
        SSL_CTX* ctx = SSL_CTX_new(DTLSv1_method());
48      if (ctx == NULL) {
                ERR_print_errors_fp(stderr);
                return NULL;
51      }

        int err = SSL_CTX_set_cipher_list(ctx, "HIGH:MEDIUM:aNULL");
54      if (err != 1) {
                fprintf(stderr, "DTLS: unable to load ciphers, exiting\n");
                exit(1);
57      }

        FILE* paramfile = fopen("dh_param_1024.pem", "r");
60      if (paramfile == NULL) {
                fprintf(stderr, "Error opening the DH file: %s\n",
                        strerror(errno));
63              return NULL;
        }
```

```
	DH* dh_1024 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
	fclose(paramfile);
	if (dh_1024 == NULL) {
		fprintf(stderr, "Error reading the DH file\n");
		return NULL;
	}

	err = SSL_CTX_set_tmp_dh(ctx, dh_1024);
	if (err != 1) {
		fprintf(stderr, "createctx: unable to set DH parameters\n");
		ERR_print_errors_fp(stderr);
		SSL_CTX_free(ctx);
		ctx = NULL;
		return NULL;
	}
	return ctx;
}

static struct dtls_peer*
dtlsnewpeer(const struct sockaddr *so, bool client)
{
	assert(so != NULL);

	static SSL_CTX *ctx = NULL;
	int err = 0;
	if (ctx == NULL) {
		ctx = createctx();
		if (ctx == NULL) {
			fprintf(stderr, "dtlsnewpeer: unable to create CTX\n");
			return NULL;
		}
	}

	struct dtls_peer p;

	err = BIO_new_bio_pair(&p.bio, MTU, &p._b2, MTU);
	if (err != 1) {
		fprintf(stderr, "dtlsnewpeer: unable to create bio pair\n");
		ERR_print_errors_fp(stderr);
		return NULL;
	}

	p.ssl = SSL_create(ctx, p._b2, client);
	if (p.ssl == NULL) {
		BIO_free(p.bio);
		BIO_free(p._b2);
		return NULL;
	}

	return addpeer(&p, so);
}

ssize_t
```

```
dtlssendto(int s, const void *msg, size_t len, int flags,
        const struct sockaddr *to, socklen_t tolen)
{
        ssize_t ret = -1;
        int err = 0;
        unsigned char* tbuf[MTU] = {0};
        ssize_t retv = -10;

        if (len > MTU) {
                /* what about TLS encapsulation size? */
                errno = EMSGSIZE;
                return -2;
        }

        struct dtls_peer *p = getpeer(to, tolen);
        if (p == NULL) {
                fprintf(stderr, "unknown peer!\n");
                p = dtlsnewpeer(to, true);
                if (p == NULL) {
                        fprintf(stderr, "dtlssendto: peer creation failed\n");
                        return -3;
                }
                err = SSL_connect(p->ssl);
                retv = 0;
        } else {
                err = SSL_write(p->ssl, msg, len);
                if (err != len) {
                        int ret = SSL_get_error(p->ssl, err);
                        fprintf(stderr, "dtlssendto: SSL_write/connect returned %d"
                                " (len was %d): error %d\n", err, len, ret);
                        ERR_print_errors_fp(stderr);
                }
                retv = len;
        }

        ret = BIO_read(p->bio, tbuf, MTU);
        if (ret <= 0) {
                if (ret < 0)
                        fprintf(stderr, "dtlssendto: BIO_read error (%d)\n",
                                ret);
                return 0;
        }

        err = sendto(s, tbuf, ret, flags, to, tolen);
        if (err != ret) {
                fprintf(stderr, "dtlssendto: sendto: %s\n",
                        strerror(errno));
                return -1;
        }
        return retv;
}

static struct dtls_peer*
```

```
socktobssl(int s, int flags)
{
174        ssize_t ret = −1;
           unsigned char∗ tbuf[MTU] = {0};
           struct sockaddr_storage ifrom;
177        socklen_t ifromlen = sizeof ifrom;
           ret = recvfrom(s, tbuf, MTU, flags, (struct sockaddr∗) &ifrom,
                          &ifromlen);
180        if (ret == −1) {
                   fprintf (stderr, ”dtlsrecvfrom: recvfrom: %s\n”,
                           strerror (errno));
183                return NULL;
           }

186        struct dtls_peer ∗p = getpeer((struct sockaddr∗) &ifrom, ifromlen);
           if (p == NULL) {
                   fprintf (stderr, ”unknown peer!\n”);
189                p = dtlsnewpeer((struct sockaddr∗) &ifrom, false);
                   if (p == NULL)
                           return NULL;
192        }

#ifdef DBG_DTLS
195        fprintf (stderr, ”Writing %d octet(s) to the bio\n”, ret);
#endif
           int nret = BIO_write(p−>bio, tbuf, ret);
198        assert (nret == ret);
           int needed = BIO_get_read_request(p−>bio);
           if (needed > 0) {
201                fprintf (stderr, ”dtlsrecvfrom: Still %d octet(s) needed... \n”,
                           needed);
                   return NULL;
204        }
           return p;
}
207
static int
ssltobuf(int s, void∗ buf, size_t len, int flags,
210         struct sockaddr ∗from, socklen_t ∗fromlen, struct dtls_peer∗ p)
{
           assert (p != NULL);
213
           int ret2 = SSL_read(p−>ssl, buf, len);
           if (ret2 < 0) {
216                int err = SSL_get_error(p−>ssl, err);
                   fprintf (stderr, ”SSL_read: returned %d, error %d\n”, ret2, err);
                   ERR_print_errors_fp(stderr);
219        }

           int retb = 0;
222        if ((retb = BIO_ctrl_pending(p−>bio)) > 0) {
#ifdef DBG_DTLS
                   fprintf (stderr, ”sending pending data (%d octets to send)\n”,
```

```
225                             retb);
#endif
                unsigned char* tbuf[MTU] = {0};
228             int ret3 = BIO_read(p->bio, tbuf, MTU);
                if (ret3 != retb)
                        fprintf(stderr, "dtlsrecvfrom: warning "
231                             "datatosend > MTU, %d != %d\n", ret3, retb);
                retb = sendto(s, tbuf, ret3, flags, (struct sockaddr*) &p->addr,
                              addrsize((struct sockaddr*) &p->addr));
234             if (retb < 0) {
                        fprintf(stderr, "sendto: %s\n", strerror(errno));
                }
237             assert(ret3 == retb);
                assert(ret2 <= 0);
                return dtlsrecvfrom(s, buf, len, flags, from, fromlen);
240     }
        assert(retb == 0);
        if (ret2 < 0)
243             return -1;

        if (from != NULL && fromlen != NULL) {
246             memcpy(from, &p->addr, sizeof(struct sockaddr_storage));
                *fromlen = addrsize(from);
        }
249     return (ssize_t) ret2;
}

252 ssize_t
dtlsrecvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from,
             socklen_t *fromlen)
255 {
        struct dtls_peer *p = NULL;
        p = socktobssl(s, flags);
258     if (p != NULL)
                return ssltobuf(s, buf, len, flags, from, fromlen, p);
        return -1;
261 }
```

On le voit, on peut séparer les fonctions définies dans ce fichier en différentes catégories :

1. Une fonction d'initialisation à usage unique, il s'agit de `createctx` qui est utilisée pour charger les paramètres de Diffie-Hellman et créer le `SSL_CTX` (comme toutes les connexions utilisent DTLSv1, il est ici partagé. Pour cela on utilise `DTLSv1_method` qui convient aussi bien pour les clients que pour les serveurs). Cette fonction est appelée une seule fois lors du premier appel à `dtlsnewpeer` : le `SSL_CTX` utilisé par cette dernière fonction étant déclaré `static`, il est conservé pour les appels ultérieurs.
2. Les fonctions d'initialisation d'un « peer » qui sont appelées lors de la création d'une nouvelle connexion. Il s'agit de `dtlsnewpeer` et `SSL_create`. La première est chargée de remplir une structure `peer_data` temporaire en initialisant ses membres. Elle crée notamment les deux BIOs qui constituent la BIO_pair permettant de récupérer la sortie de SSL (cf. figure 2), ainsi que justement, l'objet `SSL` lui-même (dont la création est déléguée à `SSL_create`). Cette fonction appelle enfin `addpeer` pour l'enregistrement à

plus long terme de cette structure.

3. La fonction `dtlssendto` qui est une longue fonction mais son découpage n'aurait présenté que peu d'intérêt. Cette fonction utilise `getpeer` pour récupérer les données correspondant à la destination. Si `getpeer` renvoie `NULL`, c'est que l'on a pas encore communiqué avec la destination, auquel cas on appelle `dtlsnewpeer` pour créer les structures nécessaires, puis (avec `SSL_connect`, `BIO_read` puis `sendto`) on envoie le premier message d'initialisation de la connexion. Si on a déjà une connexion établie, on passe alors les données pour chiffrement à SSL (`SSL_write`), puis on lit les données chiffrées à la sortie de la paire de BIOs (`BIO_read`) et on les copie sur la socket avec `sendto`.
4. Enfin, la fonction `dtlsrecvfrom`, qui a été séparée en deux sous-fonctions, `socktobssl` et `ssltobuf`. Le rôle de la première est de copier les données qui arrivent par la socket sur le BIO correspondant à la connexion SSL avec l'émetteur ou, dans le cas où celui-ci est inconnu, de créer une nouvelle connexion en supposant que le packet reçu est le premier message `HELLO` du *handshake* de DTLS. La seconde de ces fonctions a un rôle double : elle lit les données telles que déchiffrées par SSL et les copie sur le buffer passé en paramètre par l'utilisateur. Cependant, il est aussi possible que `SSL_read` renvoie 0, dans ce cas il s'agit souvent de la phase d'initialisation de la connexion et SSL a de nouveau des données à envoyer. Dans ce dernier cas, la fonction envoie ces données puis se remet en attente d'un nouveau paquet.

## 2.1 Limites

En raison de sa simplicité, le code du module tel que présenté ici souffre d'un certain nombre de défauts qui doivent être pris en compte dans un programme réel, le plus gênant étant le risque de deadlock : en effet, nous l'avons dit, `dtlssendto`, employée pour la première fois par le client, peut renvoyer 0, auquel cas il faut réessayer après avoir appelé `dtlsrecvfrom` (accessoirement comme cette dernière ne rend la main qu'après avoir lu des données, cela veut dire que l'application serveur doit envoyer en premier des données au client une fois la connexion établie). Le problème est qu'UDP n'est pas un protocole fiable, et la perte d'un paquet peut bloquer à la fois le client et le serveur dans `dtlsrecvfrom`. La solution est d'utiliser des sockets non bloquantes et `select` pour attendre que des données arrivent, avec un temps d'expiration au bout duquel on renvoie les données.

Un problème supplémentaire possible vient du fait que nous ne nous sommes pas préoccupés ici des problèmes de MTU.

Enfin, nous n'avons pas traité du problème de la fin des sessions. Contrairement à ce qui se passe avec TCP, une application UDP ne peut se baser sur la couche réseau pour déterminer quand une connexion est terminée. Il faut donc que l'application implémente un moyen de déterminer la fin des connexions (message explicite, délai d'expiration, etc.) et détruise les données relatives à la connexion au moment nécessaire. Dans le cas contraire, les données relatives à des connexions qui n'existent plus vont s'accumuler et épuiser petit à petit la mémoire.

# 3 Conclusion

Nous avons présenté sommairement dans ce document comment utiliser l'implémentation de DTLS fournie par OpenSSL dans une application pour chiffrer des communications au-

dessus d'UDP et donné un aperçu des problèmes qui se posent.

En l'absence actuelle de documentation fournie par OpenSSL concernant DTLS, ce document devrait faciliter l'usage de ce protocole, en particulier pour les personnes qui manquent de l'envie ou du temps nécessaire pour se plonger dans le code source de cette bibliothèque.

Le code source des exemples est téléchargeable sur le site web de l'auteur (`http://www.lafraze.net/nbernard/`). Celui de la bibliothèque OpenSSL l'est sur `http://www.openssl.org`.

## A Code de création de la socket

Listing 7 – `makesock`, crée une socket UDP

```
static int
makesock(int port)
{
        int sock = 0;
        struct sockaddr_in addr;
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
        addr.sin_port = htons(port);

        sock = socket(PF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                fprintf (stderr, "socket: %s\n", strerror (errno));
                exit (EXIT_FAILURE);
        }

#ifdef SERVER      /* inutile pour le client */
        if (bind(sock, (struct sockaddr*) &addr, sizeof addr)) {
                fprintf (stderr, "bind: %s\n", strerror (errno));
                exit (EXIT_FAILURE);
        }
#endif

        return sock;
}
```

## B Licence du module `dtlsplex.c`

Le lecteur désireux d'utiliser le module `dtlsplex.c` ou une version dérivée doit être conscient que celui-ci est fourni sous la licence suivante :


```
/*
 * Copyright (c) 2006 Nicolas Bernard
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
```

```
* MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
* ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
* WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
* ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
* OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*
*/
```

Nous attirons également l'attention du lecteur sur la licence d'OpenSSL, disponible à l'adresse suivante : `http://www.openssl.org/source/license.html`.

# C Quelques directions pour aller plus loin

Comme nous l'avons dit, l'implémentation de DTLS d'OpenSSL n'est actuellement pas documentée. Le code source de cette implémentation est réparti entre le fichier `crypto/bio/bss_dgram.c` et les fichiers `ssl/d1_*.c`. Un exemple de l'utilisation de DTLS est enfoui dans le code de l'application `openssl`, en particulier dans les fichiers `apps/s_client.c` et `apps/s_server.c` (correspond à une utilisation simple comme dans notre section 1).