Part2. プログラミング

はじめに

これからの各章ではTLSプロトコル、暗号アルゴリズム、公開鍵証明書・PKIなどに分けて典型的な処理に関するサンプルプログラムについて解説します。 各サンプルでは、以下の順に説明します。コマンドの使用例では、実際にサンプルプログラムを動作させてWireshakのパケットキャプチャーや生成されるファイルの内容などを使ってTLSプロトコルや暗号アルゴリズムへの理解を深めます。

- 概要
- コマンドの形式と使用例
- プログラム
- その他

「第6章：TLSプロトコル」では、TLSプロトコルを使ったクライアント、サーバーの通信の実現例を紹介します。第2章の始めに、簡単なTCPのクライアント、サーバープログラムをベースにTLS化する方法について解説しました。この章ではまず、読者が各種の実験ができるようにその簡単なクライアント、サーバーに簡単なアーギュメント処理を追加したサンプルプログラムを紹介します。また、それをベースに事前共有鍵、セッション再開などの処理に発展させた例について紹介します。また、これらのサンプルプログラムを実際に動作させ、Wiresharkによるパケットキャプチャーの内容を見ながらプロトコルへの理解も深めていきます。

ディレクトリ名	説明
01.tcp-tls	もっとも簡単なTCPクライアントとサーバ TLSクライアントとサーバ
02.client-server	簡単なクライアントとサーバ このサンプルをベースに他の機能の サンプルに展開
03.tls-ext	TLS拡張のサンプルコード
04.psk	事前共有鍵(PSK)
05.resume	セッション再開
06.early-data	Early Data (0-RTT)

「第7章：暗号アルゴリズム」では、TLSベースにもなっている各種の暗号処理のプログラム例を紹介します。アルゴリズムごとにハッシュ、メッセージ認証コード、RSA公開鍵による暗号化、署名検証、CRX509証明書の生成、検証などを紹介します。

ディレクトリ名	説明
01.hash	ハッシュ
02.hmac	HMACによるメッセージ認証コード
03.sym	共通鍵暗号(AES-CBC, AES-GCM)
04.keyGen	公開鍵生成

ディレクトリ名	説明
05.rsaEnc	RSA暗号
06.rsaSig	RSA署名
07.cert	X509証明書

「第8章：プログラム構造」では、各種のコンフィグレーションやプラットフォームの動作モードなどに対応するためのプログラム例を紹介します。

ディレクトリ名	説明
01.iocallback	ネットワークIOコールバック
02.nofilesys	ファイルシステムなし
03.nonblocking	ノンブロックネットワーキング
04.supperloop	スーパーループによるクライアント

共通事項

1) サンプルプログラム

各セッションでは、サンプルプログラムの機能概要、C言語のコード、そこで利用されるAPI、また関連する情報についてまとめます。サンプルプログラムのコードは紙面の都合でエラー処理など詳細に関しては省略した形で紹介します。エラー処理を含む実行可能なサンプルプログラムは連携サイト(...)からダウンロードすることができます。

2) プラットフォーム

TLS, 暗号処理APIについては、できる限りOpenSSL、wolfSSLで共通するAPIを使用しています。特に断りのない限りOpenSSL, wolfSSLの両者で動作します。ただし、「第8章：プログラム構造」では主にwolfSSLにおける実装例について紹介します。

サンプルプログラムはLinux, Mac OS, WindowsのC言語コンパイラーによるコマンド環境で動作するように配慮されています。

TCP以下の基本的なネットワーク環境については、各OSが提供するTCP環境、C言語環境で提供されるBSDソケットAPIを前提としています。

OSとコンパイル環境の詳細については「付録1。プログラミング環境」を参照してください。

3) ビルド方法

各サンプルプログラムのフォルダーにはMakefileが含まれていて、makeコマンドで動作可能な実行ファイルを作ることができます。ビルド方法の詳細は「Part4 付録1 プログラミング環境 1) サンプルプログラムのビルド」を参照。

4) 共通ヘッダーファイル

本書で紹介するサンプルプログラムでは下記の共通ヘッダーファイルをインクルードします。この中には各プログラムで共通に使われる定義やロジックが含まれています。

Examples/include/example_common.h

- C言語標準ライブラリのためのヘッダーファイル
- BSD Socketライブラリのためのヘッダーファイル
- TLS1.3のセッション鍵を得るためのコールバック　使い方は 「Part4 付録1 プログラミング環境 2) デバッグツール」を参照。

第6章 TLSプロトコル

はじめに

この章では、「第2章 TLSプロトコル」で説明したTLSの実装のための各種のサンプルプログラムを紹介します。

第2章の始めに、簡単なTCPのクライアント、サーバープログラムをベースにTLS化する方法について解説しました。このサンプルのクライアントは同じノードの上のローカルホストのサーバーとTCP接続をベースにTLS接続を確立し、クライアント、サーバー間の1往復のアプリケーションメッセージの通信を行うサンプルでした。このプログラムのソースコードは01.tcp-tlsに格納されています。

「6.1 クライアント・サーバー通信」では読者がもう少しさまざまな条件でTLS接続と通信を実験できるように機能を拡張しています。このサンプルでは、例えばクライアントの起動時に接続先のドメイン名、ポート番号、サーバー認証のためのCA証明書などをコマンドアーギュメントで指定できるようになっています。「6.1 クライアント・サーバー通信」では、「2.1 フルハンドシェイク」で説明したフルハンドシェイクによってTLS接続を実現しクライアント・サーバー間のアプリケーション・データの通信を実行します。

TLSでは接続に伴って各種のオプション機能をTLS拡張として指定できるようになっています。「6.2 TLS拡張」では前のサンプルの補足として、各種のTLS拡張の指定方法についてサンプルプログラムで紹介します。

「6.3 事前共有鍵」では、「6.1 クライアント・サーバー通信」のサンプルをベースに、TLS接続の形態としてフルハンドシェイクではなく、事前共有鍵によるTLS接続のサンプルプログラムを紹介します。この形態の接続には、接続ハンドシェイク中に事前共有鍵を指定するコールバックをアプリケーション側で用意する必要があります。サンプルプログラムではその方法について解説します。

TLS1.3のセッション再開ではセッションチケットを使用します。セッション再開のサンプルプログラムではTLS1.3のセッションチケットを使用したセッション再開のサンプルについて解説します。

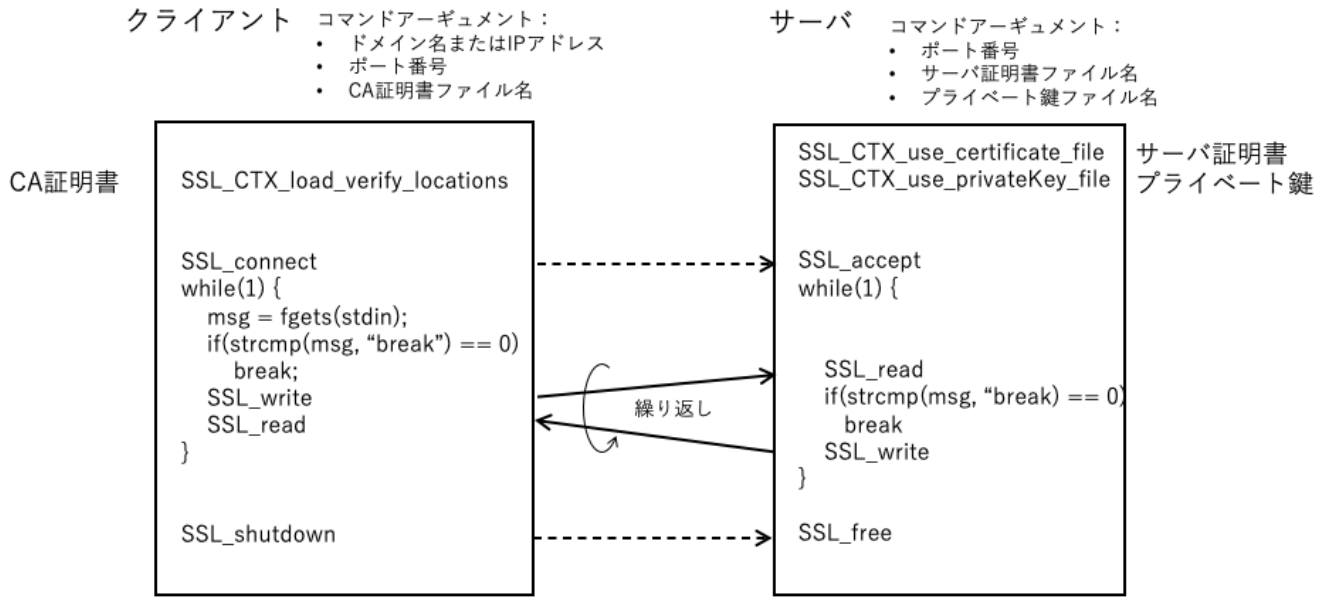
6.1 クライアント・サーバー通信

6.1.1 概要

このサンプルはクライアントとサーバーの間でTLS接続による簡単なアプリケーションメッセージ通信を行います。クライアントのコマンドアーギュメントでTLS接続先のIPアドレスまたはドメイン名、ポート番号、CA証明書のファイル名を指定します。サーバーのコマンドアーギュメントでは、接続を受け付けるポート番号、サーバー証明書のファイル名、プライベート鍵のファイル名を指定します。

クライアントはサーバーとのTLS接続を確立したあと、標準入力からのメッセージをサーバーに送信します。サーバーは受信したメッセージを標準出力に表示するとともに、所定のメッセージをクライアントに返送します。クライアントはサーバーから返却されたメッセージを標準出力に表示します。クライアントは入力メッセージがある限りこれを繰り返します。サーバーもクライアントからのメッセージがある限り、返信します。クライアントの入力メッセージが"shutodwn"の場合、クライアントはこれをサーバーに送信したあと、TLS接続を解除し処理を終了します。サーバー側も"break"を受信した場合、処理を終了します。

TLS接続の際にピア認証を行います。サンプルプログラムではクライアント側がサーバー認証を行い、サーバー側はその認証要求に応えます。そのために、クライアント側にはCA証明書、サーバー側にはサーバー証明書とプライベート鍵をあらかじめ登録しておきます。



このサンプルプログラムでは一つの接続だけを使って通信を行う例となっています。TLS接続はSSL構造体で管理しています。例えば、サーバ側で複数スレッドで同時に複数の接続を確立して通信を行いたいような場合はSSL_new関数で複数のSSL構造体を確保して使用します。

また、クライアントが異なる複数のサーバに対して接続したい場合や、一つのサーバプログラムで複数の異なるアイデンティティを持つサーバ(異なるサーバ証明書を持つサーバ)を実現したい場合があります。そのような場合にはSSL_CTX_new関数で複数の異なるコンテクスト構造体(SSL_CTX)を確保して、異なる証明書をロードして使用します。

6.1.2 コマンド形式と使用例

サーバ(Server-tls):

- 第1アーギュメント：サーバ認証のためのサーバ証明書(省略可)
- 第2アーギュメント：サーバ認証のためのプライベート鍵(省略可)
- 第3アーギュメント：ポート番号(省略可)

第3、第２、第1アーギュメントの順に省略することができます。

クライアント(Client-tls)

- 第1アーギュメント：サーバ認証のためのCA証明書(省略可)
- 第2アーギュメント：接続先IPアドレス(省略可)
- 第3アーギュメント：ポート番号(省略可)

第3、第２、第1アーギュメントの順に省略することができます。

正常接続の使用例

1) 正常ケース

サーバ用のコマンドウィンドウでサーバコマンドを起動します。クライアントからの接続要求待ちに入ります。

```
$ ./Server-tls
Waiting for a connection...
```

同一マシン上のクライアント用のコマンドウィンドウでクライアントコマンドを起動します。ローカルのサーバへのTLS接続が完了し、サーバに送信する入力プロンプトが表示されます。

```
$ ./Client-tls
Send to localhost(127.0.0.1)
Message to send:
```

サーバ側には、クライアントとTLS接続完了のメッセージが出力されます。

```
Client connected successfully
```

クライアント側でメッセージを入力するとサーバに送信され、サーバ側より送られたメッセージが表示されます。

```
Message to send:Hello server
Received: I hear ya fa shizzle!
```

クライアント側でメッセージに "break" を入力すると、サーバ側に送信したのちTLSを切断し、プログラムを終了します。

```
Message to send: break
Sending break command
End of TLS Client
```

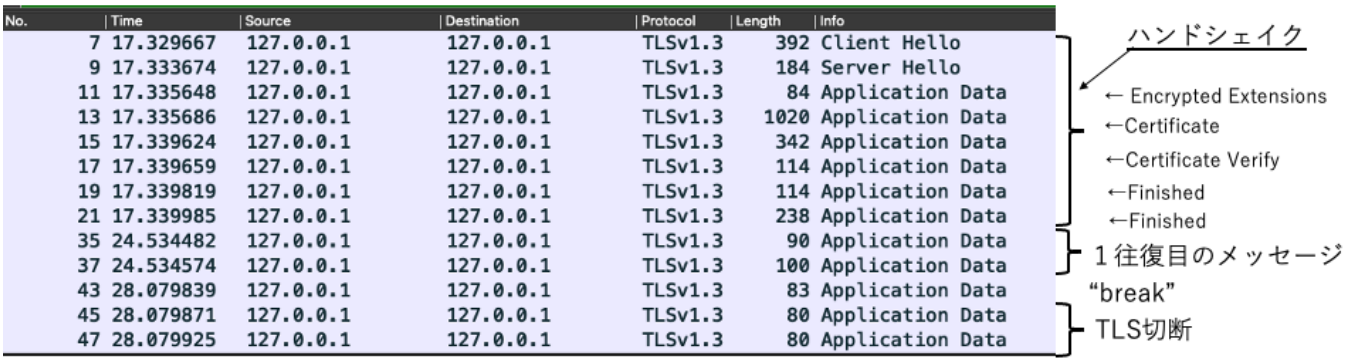
サーバ側でも "break" が受信された旨表示したのち、TLS切断し、プログラムを終了します。

```
Received: break

Received break command
Closed the connection
```

この間、WiresharkでLoocal Loopのパケットキャプチャを取得すると下のようなパケットが順次やりとりされるのを見ることができます。 TLSのハンドシェークは、冒頭のClient Hell, Server Helloの後は暗号化されているのです

べて "Application Data" レコード として表示されますが、実際にはハンドシェークの後半部分も含まれています。これらを復号するためにはセッション鍵情報が必要です。 復号の方法については付録を参照してください。



Client Hello の内容を見てみます。レコード層とClient Helloメッセージ中の二箇所にTLSのバージョン番号情報として "TLS1.2" が格納されているのがわかります。これらのフィールドは後方互換性のためのものでTLS1.3としては内容は無視されます。ネットワーク転送時の 通過点のミドルボックスの中にはTLS1.3を示す0x0304を認識できないものが残されているためにこれらのフィールドには過去バージョン値が 埋め込まれています。

```
TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 331
    Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 327
        Version: TLS 1.2 (0x0303)
```

TLS1.3では、新たなTLS拡張としてsupported_versionsが追加され下のようにこの拡張にサポートするTLSのバージョンのリストが明示されます。

```
Extension: supported_versions (len=7)
    Type: supported_versions (43)
    Length: 7
    Supported Versions length: 6
    Supported Version: TLS 1.3 (0x0304)
```

```
    Supported Version: TLS 1.2 (0x0303)
    Supported Version: TLS 1.1 (0x0302)
```

クライアントのサンプルプログラムではTLSコンテクストの確保時に下のように任意バージョン(SSLv23_client_method)を指定しているので、実際に送られる Client Helloにはビルド時に含まれているTLS1.1, 1.2, 1.3の３バージョンがリストされています。

```
ctx = SSL_CTX_new(SSLv23_client_method()))
```

Cipher Suites拡張には、利用できる暗号スイートとしてTLS1.2のものに加えてTLS1.3用として下のようなスイートが含まれています。

```
Cipher Suites (27 suites)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
```

また、Supported Groups には、(EC)DHE鍵合意で利用できる楕円曲線の種類、あるいはDH鍵長(ffdhe)がリストされています。

```
Supported Groups (5 groups)
    Supported Group: secp521r1 (0x0019)
    Supported Group: secp384r1 (0x0018)
    Supported Group: secp256r1 (0x0017)
    Supported Group: secp224r1 (0x0015)
    Supported Group: ffdhe2048 (0x0100)
```

TLS1.3ではClient Helloの中で、上のような暗号スイートや曲線種別だけでなくKey Share 拡張が追加され、(EC)DH鍵合意のためのパラメータと鍵情報も送ります。 これによって、Client Helloを受け取ったサーバ側では暗号スイートとこのKey Shareの内容に合意するならば、サーバ側の鍵情報と合わせてプリマスターキー の生成、鍵導出を行い、暗号化を開始することができるようになりました。

```
Key Share extension
    Client Key Share Length: 69
    Key Share Entry: Group: secp256r1, Key Exchange length: 65
        Group: secp256r1 (23)
        Key Exchange Length: 65
        Key Exchange: 042a9e4759a37da0cab6a1d55071d7…
```

さらに、Client Hello には、signature_algorithms拡張に署名に使用できる署名スキームの一覧が示され、この後のサーバ、クライアント認証や 鍵導出に使用されます。

```
Extension: signature_algorithms (len=32)
    Type: signature_algorithms (13)
    Length: 32
    Signature Hash Algorithms Length: 30
    Signature Hash Algorithms (15 algorithms)
        Signature Algorithm: ecdsa_secp521r1_sha512 (0x0603)
            Signature Hash Algorithm Hash: SHA512 (6)
            Signature Hash Algorithm Signature: ECDSA (3)
        Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
            Signature Hash Algorithm Hash: SHA384 (5)
            Signature Hash Algorithm Signature: ECDSA (3)
        Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
            Signature Hash Algorithm Hash: SHA256 (4)
            Signature Hash Algorithm Signature: ECDSA (3)
        ...
```

一方、サーバから送られるServer Helloにはサーバの合意したTLSのバージョンや暗号スイートとともにKey_Share拡張に(EC)DHの鍵情報が格納されています。 クライアント側ではこれを使用してサーバ側と同様にプリマスターキーの生成、鍵導出を行い暗号化を開始します。

```
    Handshake Protocol: Server Hello
        ...
        Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
        ...
        Extension: key_share (len=69)
            Type: key_share (51)
            Length: 69
            Key Share extension
                Key Share Entry: Group: secp256r1, Key Exchange length: 65
                    Group: secp256r1 (23)
                    Key Exchange Length: 65
                    Key Exchange:
04b09ee9645d87359a4b6729be30c95fad6dda7a660052493d134f0b0740e01bf1c4b1be…
        Extension: supported_versions (len=2)
            Type: supported_versions (43)
            Length: 2
            Supported Version: TLS 1.3 (0x0304)
```

サーバからはこの後、Encrypted ExtentionとしてServer Hello に付帯するTLS拡張で暗号化されたものが送られますが、現在のところ この部分にはあまり重要な情報は含まれていません。

次に、サーバからはサーバ認証のためのサーバ証明書チェーン(Certificate)と検証用の署名(Certificate Verify)が送られますが、これらの部分は暗号化されいて通常は見ることができません。CertificateメッセージにはDER形式の証明書が格納されます。

```
Handshake Protocol: Certificate
    Handshake Type: Certificate (11)
    Length: 938
```

```
    Certificate Request Context Length: 0
    Certificates Length: 934
    Certificates (934 bytes)
        Certificate Length: 929
        Certificate: 3082039d30820285020101300d0 … (pkcs-9-at-
emailAddress=info@,,,com,id-at-commonName=www.wolfssl.com,id-at-
organizationalUnitName=Support,id-at-organizationName=WolfSSL Japan,id
            signedCertificate
                serialNumber: 0x01
                signature (sha256WithRSAEncryption)
                    Algorithm Id: 1.2.840.113549.1.1.11
(sha256WithRSAEncryption)
                issuer: rdnSequence (0)
            ...

            algorithmIdentifier (sha256WithRSAEncryption)
                Algorithm Id: 1.2.840.113549.1.1.11
(sha256WithRSAEncryption)
            Padding: 0
            encrypted: 5ecc342d4d3fb775600e5e3039da737d0ef7e…
        Extensions Length: 0
```

次のCertificate Verifyメッセージでは署名検証用の署名が送られ、クライアントがCertificateで送られた証明書を使ってこれを検証します。 メッセージには、署名に使用した署名スキーム(Signature Algorithm)と署名(Signature)が格納されています。

```
TLSv1.3 Record Layer: Handshake Protocol: Certificate Verify
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 281
    [Content Type: Handshake (22)]
    Handshake Protocol: Certificate Verify
        Handshake Type: Certificate Verify (15)
        Length: 260
        Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
            Signature Hash Algorithm Hash: Unknown (8)
            Signature Hash Algorithm Signature: SM2 (4)
        Signature length: 256
        Signature: 28462795dabf4f86da81b912755a775b850509eaf9bf14…
```

このサンプルプログラムでは、サーバ認証のみを行うようになっていますが、クライアント認証も行う場合(相互認証)はクライアント側からも 同様の証明書と署名を送り、サーバ側がそれを検証します。

これらのメッセージの後、双方がハンドシェークの終了を示す"Finished"メッセージを送りハンドシェークを終了します。

2) デバッグログの利用

TLS1.3ではハンドシェイクの様子は暗号化されているために十分確認することが難しい面があります。セッション鍵を特別な方法で取得して、キャプチャー内容を復号することも可能ですがセキュリティの観点からあまり安易に利用すべきではありません。

この様子をwolfSSLのデバッグログ機能を利用して見ることができます。ライブラリーをビルドする際に "--enable-debug"オプションを有効化するとライブラリー内の実行状況を逐次でバック情報として出力することができます(付録1参照)。プロトコルの状態遷移は"connect state:"のプリフィックスで出力されます。また、関数の呼び出しは"wolfSSL Entering"のプリフィックスです。ハンドシェイク中の各メッセージ処理の関数名は、

- 送信処理: "SendTLS13"
- 受信処理："DoTLS13"

のプリフィックスで命名されています。それらのフィルターでログを抽出するとクライアント側では次のようなログを得ることができます。

```
ライブラリ初期化、コンテクスト確保：
wolfSSL Entering wolfSSL_Init
wolfSSL Entering wolfSSL_CTX_new_ex
wolfSSL Entering wolfSSL_CertManagerNew
wolfSSL Leaving WOLFSSL_CTX_new, return 0

証明書ロード、TLS接続準備：
wolfSSL_CTX_load_verify_locations_ex
wolfSSL_CTX_load_verify_locations_ex

wolfSSL Entering SSL_new
wolfSSL Leaving SSL_new, return 0
wolfSSL Entering SSL_set_fd
wolfSSL Entering SSL_set_read_fd
wolfSSL Leaving SSL_set_read_fd, return 1
wolfSSL Entering SSL_set_write_fd
wolfSSL Leaving SSL_set_write_fd, return 1


TLS接続：
wolfSSL Entering SSL_connect()
wolfSSL Entering SendTls13ClientHello
connect state: CLIENT_HELLO_SENT
wolfSSL Entering DoTls13ServerHello
wolfSSL Entering wolfSSL_connect_TLSv13()
connect state: HELLO_AGAIN
connect state: HELLO_AGAIN_REPLY
wolfSSL Entering DoTls13EncryptedExtensions
wolfSSL Entering DoTls13Certificate
wolfSSL Entering DoTls13CertificateVerify
wolfSSL Entering DoTls13Finished
connect state: FIRST_REPLY_DONE
connect state: FIRST_REPLY_FIRST
connect state: FIRST_REPLY_SECOND
wolfSSL Entering SendTls13Certificate
connect state: FIRST_REPLY_THIRD
wolfSSL Entering SendTls13CertificateVerify
```

```
connect state: FIRST_REPLY_FOURTH
wolfSSL Entering SendTls13Finished
connect state: FINISHED_DONE
wolfSSL Leaving wolfSSL_connect_TLSv13(), return 1

アプリケーションメッセージ通信：
wolfSSL Entering SSL_write()
wolfSSL Leaving SSL_write(), return 14
wolfSSL Entering wolfSSL_read()
wolfSSL Entering wolfSSL_read_internal()
wolfSSL Leaving wolfSSL_read_internal(), return 22


切断、リソース解放：
wolfSSL Entering SSL_shutdown()
wolfSSL Leaving SSL_shutdown(), return 2
wolfSSL Entering SSL_free
wolfSSL Leaving SSL_free, return 0
wolfSSL Entering SSL_CTX_free
wolfSSL Entering wolfSSL_CertManagerFree
wolfSSL Leaving SSL_CTX_free, return 0
wolfSSL Entering wolfSSL_Cleanup
```

3) TLSバージョンを変えてみる

サンプルプログラムのサーバとクライアントは、コンテクスト確保時に受け付けるTLSバージョンとして、ライブラリビルド時に組み込まれたすべてのバージョンを 受け付ける指定になっています。

サーバ側：
ctx = SSL_CTX_new(SSLv23_server_method())

クライアント側：
ctx = SSL_CTX_new(SSLv23_client_method()))

ここで、クライアント側をTLS1.3のみ受け付ける指定に変更してみます。

ctx = SSL_CTX_new(TLSv1_3_client_method()))

Client Hello のsupported_versionsにはTLS1.3のみがリストされていることがわかります。

```
Extension: supported_versions (len=3)
    Type: supported_versions (43)
    Length: 3
    Supported Versions length: 2
    Supported Version: TLS 1.3 (0x0304)
```

次に、クライアント側をTLS1.2のみ受け付ける指定に変更してみます。

ctx = SSL_CTX_new(TLSv1_2_client_method()))

今度はTLS1.2で接続されていることが確認できます。この時、Client Hello の内容をみるとsupported_versions拡張は存在せず、 Client HelloはTLS1.2の形式であることが確認できます。

No.	Time	Source	Destination	Protocol	Length	Info
5	0.000216	127.0.0.1	127.0.0.1	TLSv1.2	220	Client Hello
7	0.000293	127.0.0.1	127.0.0.1	TLSv1.2	151	Server Hello
9	0.000318	127.0.0.1	127.0.0.1	TLSv1.2	1000	Certificate
11	0.005711	127.0.0.1	127.0.0.1	TLSv1.2	394	Server Key Exchange
13	0.005732	127.0.0.1	127.0.0.1	TLSv1.2	65	Server Hello Done
15	0.008070	127.0.0.1	127.0.0.1	TLSv1.2	131	Client Key Exchange
17	0.008123	127.0.0.1	127.0.0.1	TLSv1.2	62	Change Cipher Spec
19	0.008152	127.0.0.1	127.0.0.1	TLSv1.2	101	Encrypted Handshake Message
21	0.008914	127.0.0.1	127.0.0.1	TLSv1.2	62	Change Cipher Spec
23	0.008933	127.0.0.1	127.0.0.1	TLSv1.2	101	Encrypted Handshake Message

次に、サーバ側をTLS1.3のみ受付るように変更します。

ctx = SSL_CTX_new(TLS1_3_server_method())

この状態で、サーバとクライアントを起動すると、サーバ側では次のようにバージョンエラーを検出します。

```
$ ./Server-tls
Waiting for a connection...
ERROR: failed SSL accept (err -326, record layer version error)
End of TLS Server
```

一方、クライアント側ではアラートメッセージを受信した旨のエラーメッセージが表示されます。

```
$ ./client-tls
Send to localhost(127.0.0.1)
ERROR: failed SSL connect (err -313, received alert fatal error)
End of TLS Client
```

このようにTLSでは通信の一方でエラーを検出した場合、相手方には単にアラートレコードを送信するだけでエラーの原因などは示さずに 通信を切断します。エラーが何らかのセキュリティに対する攻撃によるものであった場合に相手側にヒントとなりうる 情報を一切送らないようにするために、エラー時の動作はそのように規定されています。

4) サーバ認証エラー

ここでは、クライアント・サーバのサンプルプログラムを使ってサーバ認証エラーの場合の動作を見てみます。certsディレクトリーの 下のサーバ証明書をローカルにコピーして、その内容を一部修正して不正な証明書を作ります。もとの証明書と修正した証明書の内容を OpenSSL x509サブコマンドの "-text" オプションでテキスト編集したものを出力させます。diffコマンドなどで差分をみて、 修正箇所がASN1の構文を崩していないこと、署名など適当な部分が修正されていることを確認します。

```
$ cp ../../certs/tb-server-cert.pem ./tb-server-cert2.pem

テキストエディタにて修正

$ openssl x509 -in tb-server-cert.pem -text > tb-server-cert.txt
$ openssl x509 -in tb-server-cert2.pem -text > tb-server-cert2.txt
$ diff tb-server-cert.txt tb-server-cert2.txt
45c45
<          74:62:d8:6d:21:11:eb:0c:82:50:22:a0:c3:88:52:7c:b3:c4:
---
>          74:62:d8:6d:21:11:eb:0c:82:50:22:a4:c3:88:52:7c:b3:c4:
69c69
< oMOIUnyzxOk4dRH+SkcmN8pW17Wp2WbS45BiHjVtgrAALMTv2dJpk8mQUjYQTTyF
---
> pMOIUnyzxOk4dRH+SkcmN8pW17Wp2WbS45BiHjVtgrAALMTv2dJpk8mQUjYQTTyF
```

サーバウィンドウで修正した証明書を指定してサーバを起動します。

```
$ ./Server-tls tb-server-cert2.pem
Waiting for a connection...
```

クライアントウィンドウにてクライアントを起動するとサーバ認証エラーで接続に失敗します。

```
$ ./client-tls
Send to localhost(127.0.0.1)
ERROR: failed SSL connect (err -155, ASN sig error, confirm failure)
End of TLS Client
```

サーバ側にもSSL_accept中にクライアントからアラートレコードが送られてきた旨、メッセージが表示され終了します。

```
ERROR: failed SSL accept (err -313, received alert fatal error)
End of TLS Server
```

6.1.3 プログラム

1. クライアント

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

#define 定数定義

int main(int argc, char **argv)
{
```

```
    ソケット用変数, メッセージ用変数の定義

    SSL_CTX* ctx = NULL;    /* SSLコンテクスト */
    SSL*     ssl = NULL;    /* SSLオブジェクト */

    ...
    コマンドアーギュメント処理
    ...

    /* ライブラリの初期化  */
        if (SSL_library_init() != SSL_SUCCESS)
    { エラー処理  }

    /*  SSLコンテクストの確保し、CA証明書をロード  */
        if ((ctx = SSL_CTX_new(SSLv23_client_method())) == NULL)
    { エラー処理  }

        if ((ret = SSL_CTX_load_verify_locations(ctx, CA_CERT_FILE, NULL))
!= SSL_SUCCESS)
    { エラー処理  }

    ...
    TCPソケットの確保、サーバーにTCP接続
    ...

    /* SSLオブジェクトの生成、ソケットをアタッチ、サーバーにSSL接続 */
        if ((ssl = SSL_new(ctx)) == NULL)
    { エラー処理  }

        if ((ret = SSL_set_fd(ssl, sockfd)) != SSL_SUCCESS)
    { エラー処理  }

        if ((ret = SSL_connect(ssl)) != SSL_SUCCESS)
    { エラー処理  }

    /* アプリケーション層のメッセージング  */
    while (1) {
        printf("Message to send: ");
            if(fgets(msg, sizeof(msg), stdin) < 0)
            break;
            if (strcmp(msg, "\n") == 0){ /* 空行の場合、固定のHTTPリクエストを送
信 */
            strncpy(msg, kHttpGetMsg, sizeof(msg));
        } else
            msg[strnlen(msg, sizeof(msg))-1] = '\0';

        /* メッセージ送信*/
            if ((ret = SSL_write(ssl, msg, strnlen(msg, sizeof(msg)))) <
0)
        { エラー処理  }

        /* "break" なら終了 */
            if (strcmp(msg, "break") == 0) {
            printf("Sending break command\n");
```

```
            ret = SSL_SUCCESS;
            break;
        }

        /* rメッセージ受信  */
            if ((ret = SSL_read(ssl, msg, sizeof(msg) - 1)) < 0)
        { エラー処理  }

        msg[ret] = '\0';
        printf("Received: %s\n", msg);
    }

cleanup:
    ...
    リソースの解放
    ...
}
```

---

2. サーバー

```
#include <openssl/ssl.h>
#define 定数定義

int main(int argc, char **argv)
{
    ソケット用変数, メッセージ用変数の定義
    SSL_CTX* ctx = NULL;    /* SSLコンテクスト */
    SSL*     ssl = NULL;    /* SSLオブジェクト */

    ...
    コマンドアーギュメント処理
    ...

    /* ライブラリの初期化  */

    /*  SSLコンテクストの確保し、サーバー証明書、プライベート鍵をロード  */
        if (SSL_library_init() != SSL_SUCCESS)
    { エラー処理  }

        if ((ctx = SSL_CTX_new(SSLv23_server_method())) == NULL)
    { エラー処理  }

        if ((ret = SSL_CTX_use_certificate_file(ctx, server_cert,
        SSL_FILETYPE_PEM)) != SSL_SUCCESS)
    { エラー処理  }

        if ((ret = SSL_CTX_use_PrivateKey_file(ctx, server_key,
        SSL_FILETYPE_PEM)) != SSL_SUCCESS)
    { エラー処理  }

    ...
```

```
    TCPソケットの確保、bind, listen
    ...

    while(1) {
        connd = accept() /* TCP アクセプト☆ */
        { エラー処理  }

        /* SSLオブジェクトの生成、ソケットをアタッチ、アクセプト☆ */
            if ((ssl = SSL_new(ctx)) == NULL)
        { エラー処理  }

        SSL_set_fd(ssl, connd);

            if ((ret = SSL_accept(ssl)) != SSL_SUCCESS)
        { エラー処理  }

        /* アプリケーション層のメッセージング  */
        while (1) {
            /* メッセージ受信 */
                if ((ret = SSL_read(ssl, buff, sizeof(buff)-1)) <= 0)
            { エラー処理  }

            buff[ret] = '\0';
            printf("Received: %s\n", buff);

            /* "break" ならば切断 */
                if (strcmp(buff, "break") == 0) {
                printf("Received break command\n");
                break;
            }

            /* メッセージ送信 */
                if ((ret = SSL_write(ssl, reply, sizeof(reply))) < 0)
            { エラー処理  }
        }
    }
cleanup:
    ...
    リソースの解放
    ...
}
```

プログラムの説明：

1) ヘッダーファイル

#include "openssl/ssl.h": TLSプログラムで使用するAPI、データタイプなどの定義が含まれています

2) 管理構造体とポインタ

- SSL_CTX *ctx;
  一連のTLS接続処理(コンテクスト)を管理するための構造体です。同じサーバーへのTLS接続のような類似の

条件での複数のTLS接続を1つのコンテクストとして管理します。

- SSL *ssl;
  1つのTLS接続を管理するための構造体です。

- 構造体の確保と解放

  - 確保：SSL_CTX_new(), SSL_new()
  - 解放：SS*_CTX_free(), SSL_free()

- 関連情報
  SSL_CTXコンテクストにひも付けられる主な情報としては以下のようなものがあります。

  - TLSバージョン:
    コンテクストの確保時、SSL_CTX_newのアーギュメントでTLS接続時のプロトコルバージョンを指定します。 (表6.1.1 SSL_CTX_new メソッド, 表6.1.2 TLSバージョン指定関連の主なAPI参照)
  - ピア認証：
    認証のためのCA証明書、自ノードの証明書、プライベート鍵などを接続前にロードしておきます(表6.1.3 ピア認証関連のAPI参照)。
  - TLS接続に使用するソケット
    SSL_set_fd関数でTLS接続に使用するソケットをSSLにひも付けます。

3) 主なAPI

- SSL_CTX_load_verify_locations
  この例では、サーバー認証のためにクライアント側でCA証明書をTLSコンテクストにロードします。クライアント認証のためにサーバー側でも使用します。(関連APIは表6.1.3 ピア認証関連のAPIを参照)

- SSL_CTX_use_certificate_file
  この例では、サーバー認証のためにサーバー側でサーバー証明書をTLSコンテクストにロードします。クライアント認証のためにクライアント側でも使用します。(関連APIは表6.1.3 ピア認証関連のAPIを参照)

- SSL_CTX_use_privateKey_file
  この例では、サーバー認証のためにサーバー側でプライベート鍵をTLSコンテクストにロードします。クライアント認証のためにクライアント側でも使用します。(関連APIは表6.1.3 ピア認証関連のAPIを参照)

- SSL_connect
  クライアントからサーバーにTLS接続を要求するAPIです。サーバーとのTCP接続が完了している状態で、SSL_newで確保したSSLを指定してこのAPIで接続を要求します。TLSバージョンや暗号スイートの合意、サーバー認証などのハンドシェイクを行います。すべての処理が正常に完了するとこのAPIは正常終了を返却します。

- SSL_accept
  クライアントからのTLS接続要求を受け付けるAPIです。クライアントからのTCP接続要求で接続が完了している状態で、SSL_newで確保したSSLを指定してこのAPIで接続要求を受け付けます。TLSバージョンや暗号スイートの合意、必要ならばクライアント認証などのハンドシェイクを行います。すべての処理が正常に完了するとこのAPIは正常終了を返却します。

- SSL_write
  接続の相手方に対して指定された長さのアプリケーションメッセージを暗号化し送信します。正常に送信が完了した場合、指定したメッセージ長と同じ値を返却します。

- SSL_read, SSL_pending
  接続の相手方から指定された最大長以下のアプリケーションメッセージを受信しバッファーに復号し、格納します。正常に受信が完了した場合、受信したメッセージのバイト数を返却します。SSL_pendingは現在ペンディングとなっている受信メッセージのバイト数を返却します。SSL_readではこのバイト数分のメッセージをノンブロッキングで読み出すことができます。

4) 処理の流れ

クライアント

- ライブラリ初期化
  プログラムの冒頭でSL_library_init()を呼び出しライブラリを初期化します。

- TLSコンテクストの確保
  SSL_CTX_newでコンテクストを1つ確保します。このとき、接続に使用するTLSバージョンを指定します(表6.1.1 SSL_CTX_new メソッド参照)。また、サーバー認証のためのCA証明書をロードします。

- ソケット確保とTCP接続
  socket、connectによってソケットの確保とサーバーとのTCP接続を要求します。

- SSLの確保とTLS接続要求
  SSL_newでSSL接続管理の構造体を確保します。SSL_set_fdでソケットをSSLにひも付けます。SSL_connectでTLS接続を要求します。

- アプリケーションメッセージ
  SSL_write、SSL_readで、アプリケーションメッセージの送信、受信を行ます。

- 切断とリソースの解放
  TLSとTCPの切断、リソースを解放します。確保したときの逆の順序で、TLS切断とSSLの解放、ソケットの解放、コンテクストの解放の順序で実行します。

サーバー

サーバー側もクライアント側とほぼ同様の処理の流れとなります。以下、クライアント側と異なる部分を説明します。

- TLSコンテクストの確保
  サーバー認証要求を受ける側となるので、サーバー証明書、プライベート鍵をロードします。

- TCP, TLS接続
  接続要求を受け付ける側となるので、listen、acceptおよびSSL_acceptを呼び出します。

- アプリケーションメッセージ
  クライアント側と同様に、SSL_read, SSL_writeを呼び出しますが、送受信が逆順となります。

6.1.4 その他の注意点

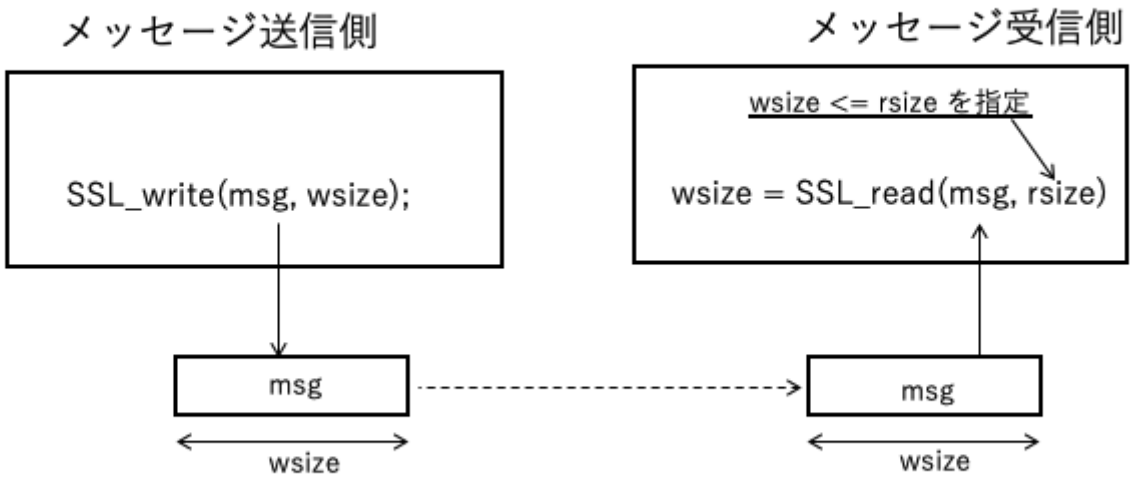
SSL_write、SSL_readで通信するメッセージは以下のような対応関係が維持されます。

TLSレコードはデフォルトでは最大16kバイトです。1回のSSL_write呼び出しによるメッセージが最大レコードサイズ以下の場合は1つのTLSレコードとして送信されます。

SSL_readで指定するメッセージ長が受信したTLSレコードと同じか長い場合は1回の呼び出しで1つのメッセージとして受信されます。送られてきたTLSレコードのサイズのほうがSSL_readで指定したメッセージサイズより長い場合には、残った分は次のSSL_read呼び出しで受信されます。

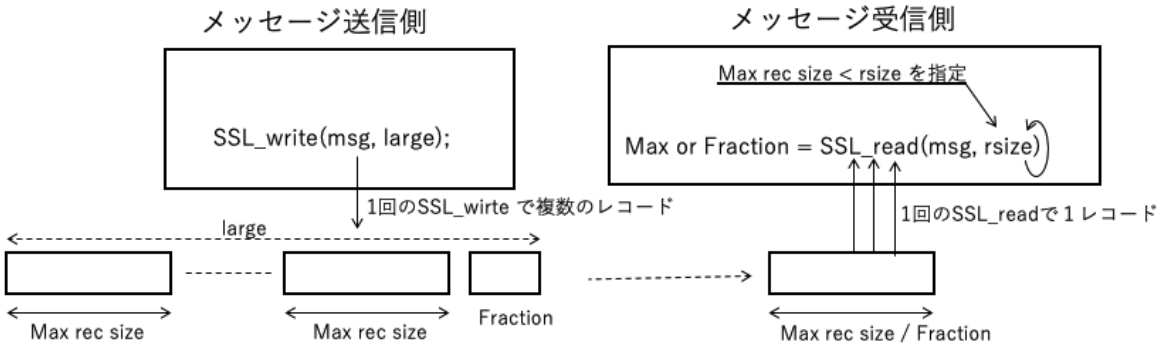
一方、SSL_readの指定サイズが長い場合でも、1回の呼び出しで受信するのは1つのTLSレコードです。関数返却値としてそのサイズを返却します。1回のSSL_read呼び出しで複数のレコードをまとめて受信することはありません。

SSL_wirteで16kバイトを超えるメッセージを指定した場合は、メッセージを16kバイト X n のレコードと残り分のメッセージのレコードに分割して複数のレコードに分割して送信します。これに対して、SSL_readは1回のAPI呼び出しに対して1レコードを読み込みます。したがって、メッセージ全体を受信するためには複数回SSL_readを呼び出す必要があります。



最大TLSレコードはデフォルトでは16kバイトですが、 MAX Fragmentを指定してしてTLSレコードの最大サイズに小さいサイズを指定した場合は、上記のレコードサイズもそのサイズとなります。

SSL_CTX_set_modeでSSL_MODE_ENABLE_PARTIAL_WRITEが指定されている場合は、SSL_writeは送信処理の状況によってメッセージ全体が送信できない場合、一部だけ送信した段階でそのバイト数を返却します。



6.1.5 参照

分類	名前	説明
サーバー	SSLv23_server_method	両者のサポートする最も高いバージョンで接続
	TLSv1_3_server_method	TLS 1.3で接続
	TLSv1_2_server_method	TLS 1.2で接続
	TLSv1_1_server_method	TLS 1.1で接続
	TLSv1_server_method	TLS 1.0で接続
クライアント	SSLv23_client_method	両者のサポートする最も高いバージョンで接続
	TLSv1_3_client_method	TLS 1.3で接続
	TLSv1_2_client_method	TLS 1.2で接続
	TLSv1_1_client_method	TLS 1.1で接続
	TLSv1_client_method	TLS 1.0で接続
サーバー/クライアント	SSLv23_method	両者のサポートする最も高いバージョンで接続
	TLSv1_3_method	TLS 1.3で接続
	TLSv1_2_method	TLS 1.2で接続
	TLSv1_1_method	TLS 1.1で接続
	TLSv1_method	TLS 1.0で接続

表6.1.1 SSL_CTX_new メソッド

分類	名前	説明

分類	名前	説明
設定	SSL_CTX_set_min_proto_version	使用する最も低いプロトコルバージョンを指定
	SSL_CTX_set_max_proto_version	使用する最も高いプロトコルバージョンを指定
参照	SSL_CTX_get_min_proto_version	設定済み最も低いプロトコルバージョンを参照
	SSL_CTX_get_man_proto_version	設定済み最も高いプロトコルバージョンを参照

表6.1.2 TLSバージョン指定関連の主なAPI

役割	機能	指定単位	ファイルシステムあり	ファイルシステムなし
認証する側	CA証明書のロード	コンテクスト	SSL_CTX_load_verify_locations	SSL_CTX_load_verify_buffer
	検証動作の指定	コンテクスト	SSL_CTX_set_verify	SSL_CTX_set_verify
	証明書チェーンの深さ指定	コンテクスト	SSL_CTX_set_verify_depth	SSL_CTX_set_verify_depth
認証される側	ノード証明書のロード	コンテクスト	SSL_CTX_use_certificate_file	SSL_CTX_use_certificate_buffer
		セッション	SSL_use_certificate_file	SSL_use_certificate_buffer
	プライベート鍵のロード	コンテクスト	SSL_CTX_use_privateKey_file	SSL_CTX_use_privateKey_buffer
		セッション	SSL_use_privateKey_file	SSL_use_privateKey_buffer

表6.1.3 ピア認証関連のAPI

6.2 TLS拡張

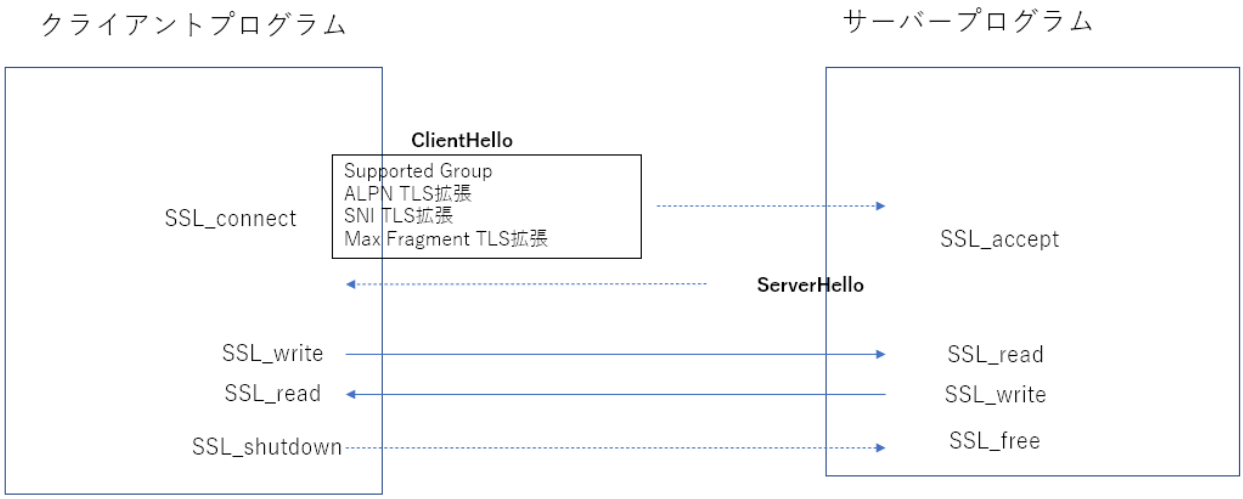
6.2.1 概要

このサンプルでは、ClientHelloレコードのTLS拡張プログラムしTLSによるメッセージ通信を行います。TLS拡張部分の設定以外は、クライアント・サンプルプログラムと同様です。

サンプルでは、以下のTLS拡張を設定します。

TLS拡張種別	設定値
Supported Versions	最小プロトコル TLS1.2 最大プロトコル TLS1.3
Supported Group	P-521, P-384 及び P-256
Max Fragment Length	1024
SNI(Server Name Indication)	localhost
Signature Algorithm	RSA+SHA256 RSA+SHA384 RSA-PSS+SHA256

またクライアントは使用するCipher スイートとして`TLS13-AES128-GCM-SHA256`を指定します。 サーバーはクライアントの要求に対して受け入れ可能な場合、ServerHelloを返信し必要に応じてTLS拡張を含めます。



6.2.2 コマンド形式と使用例

コマンド形式は「6.1 クライアントサーバ間通信」のクライアントと同じです。

Wiresharkを起動して、サーバ、クライアント双方のコマンドを起動します。

サーバ側:
$ ./Server-tls-exts

クライアント側:
$ ./Client-tls-exts

Client Hello メッセージの内容を確認します。

supported_versions拡張にはTLS1.3, 1.2だけが含まれています。

```
    Extension: supported_versions (len=5)
        Type: supported_versions (43)
        Length: 5
        Supported Versions length: 4
        Supported Version: TLS 1.3 (0x0304)
        Supported Version: TLS 1.2 (0x0303)
```

supported_groups 拡張にはP-521, P-384 及び P-256が含まれています。

```
    Extension: supported_groups (len=8)
        Type: supported_groups (10)
        Length: 8
        Supported Groups List Length: 6
        Supported Groups (3 groups)
            Supported Group: secp521r1 (0x0019)
            Supported Group: secp384r1 (0x0018)
            Supported Group: secp256r1 (0x0017)
```

max_fragment_lengthは1024となっています。

```
    Extension: max_fragment_length (len=1)
        Type: max_fragment_length (1)
        Length: 1
        Maximum Fragment Length: 1024 (2)
```

server_name拡張には指定した"localhost"がセットされています。

```
Extension: server_name (len=14)
    Type: server_name (0)
    Length: 14
    Server Name Indication extension
        Server Name list length: 12
        Server Name Type: host_name (0)
        Server Name length: 9
        Server Name: localhost
```

Signature Algorithm拡張には RSA+SHA256 RSA+SHA384 RSA-PSS+SHA256のアルゴリズムでパディングPKCS1.4とPSSだけが含まれています。

```
    Extension: signature_algorithms (len=10)
        Type: signature_algorithms (13)
        Length: 10
        Signature Hash Algorithms Length: 8
        Signature Hash Algorithms (4 algorithms)
            Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
                Signature Hash Algorithm Hash: SHA256 (4)
                Signature Hash Algorithm Signature: RSA (1)
            Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
                Signature Hash Algorithm Hash: SHA384 (5)
                Signature Hash Algorithm Signature: RSA (1)
            Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
                Signature Hash Algorithm Hash: Unknown (8)
                Signature Hash Algorithm Signature: SM2 (4)
            Signature Algorithm: rsa_pss_pss_sha256 (0x0809)
                Signature Hash Algorithm Hash: Unknown (8)
                Signature Hash Algorithm Signature: Unknown (9)
```

Cipher Suites拡張にはTLS_AES_128_GCM_SHA256のみが含まれています。

```
Cipher Suites (1 suite)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
```

6.2.3 プログラム

```
int main(int argc, char **argv)
{

    ...
    コマンドアーギュメント処理、ライブラリー初期化(6.1 クライアントに同じ)
    ...

    /*  SSLコンテクストの確保し、CA証明書をロード  */
        if((ctx = SSL_CTX_new(SSLv23_client_method())) == NULL)
    { エラー処理  }
        if ((ret = SSL_CTX_load_verify_locations(ctx, ca_cert, NULL)) !=
SSL_SUCCESS)
    { エラー処理  }

    /* TLS1.3を最大 protocol version に指定する */
        if ((ret = SSL_CTX_set_max_proto_version(ctx, TLS1_3_VERSION)) !=
SSL_SUCCESS)
    { エラー処理  }

    /* TLS1.2を最小 protocol version に指定する */
        if ((ret = SSL_CTX_set_min_proto_version(ctx, TLS1_2_VERSION)) !=
SSL_SUCCESS)
    { エラー処理  }
```

```
    /* cipher suites の指定 (TLS13-AES128-GCM-SHA256)*/
        if ((ret = SSL_CTX_set_cipher_list(ctx, CIPHER_LIST)) !=
SSL_SUCCESS)
    { エラー処理  }

    /* supported group TLS 拡張の設定 */
        if ((ret = SSL_CTX_set1_groups_list(ctx, "P-521:P-384:P-256")) !=
SSL_SUCCESS)
    { エラー処理  }

    /* signature algorithm TLS 拡張設定 */
        if ((ret = SSL_CTX_set1_sigalgs_list(ctx,
"RSA+SHA256:RSA+SHA384:RSA-PSS+SHA256")) != SSL_SUCCESS)
    { エラー処理  }

    ...
    TCP接続、TLS接続オブジェクト確保（6.1 クライアントに同じ）
    ...

    /* SNI TLS拡張を設定 */
        if ((ret = SSL_set_tlsext_host_name(ssl, "localhost")) !=
SSL_SUCCESS)
    { エラー処理  }

    /* Max Fragment Length TLS拡張を設定 */
        if ((ret = SSL_set_tlsext_max_fragment_length(ssl,
TLSEXT_max_fragment_length_1024)) != SSL_SUCCESS)
    { エラー処理  }


    以下、6.1 クライアントに同じ
    ...

cleanup:
    リソースの解放
}
```

6.2.4 TLS拡張に関する主なAPI

TLS拡張種別	関数名	説明
Supported Versions	SSL_CTX_set_max_proto_version SSL_CTX_set_min_proto_version	使用可能な最大TLSプロトコルバージョンを指定する 使用可能な最小TLSプロトコルバージョンを指定する
Supported group	SSL_CTX_set1_groups_list	サポートする楕円曲線暗号の曲線リストを指定。 複数指定する場合は、":" を区切り文字として使用。

TLS拡張種別	関数名	説明
Signature algorithm	SSL_CTX_set1_sigalgs_list	サポートする署名アルゴリズムの組み合わせを指定。 公開鍵アルゴリズムとハッシュアルゴリズムの組み合わせを"+"で結合し指定するか、 rsa_pss_pss_sha256のような表記を使用。 複数指定する場合は、":" を区切り文字として使用。
SNI	SSL_set_tlsext_host_name	ホスト名を指定します。
ALPN	SSL_set_alpn_protos	ALPNで使用するプロトコルを指定します。

6.3 事前共有鍵(PSK)

6.3.1 概要

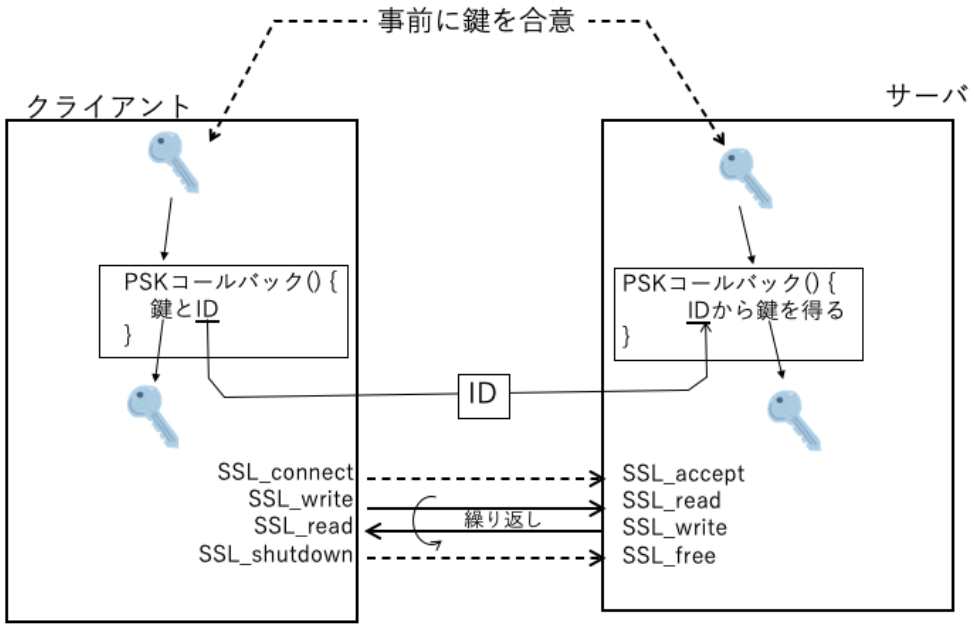
このサンプルでは、事前共有鍵によるTLS接続を行いTLSによるメッセージ通信を行います。TLS接続後のアプリケーションメッセージの通信部分は「6.1 クライアント・サーバー通信」のサンプルプログラムと同じです。

PSKによる接続では、TLS通信とは別途の何らかの方法で利用する共通の鍵値を共有しておきます。複数の鍵を利用することができます。その場合は、鍵毎にアイデンティティも取り決めておきます。

接続要求するクライアントでSSL_connectが呼ばれるとあらかじめ登録してあるコールバック関数が呼び出されるので、その中でこの接続で使用するアイデンティティと鍵値をアーギュメントに返します。受け取ったライブラリーでは鍵値はライブラリー内に保存し、アイデンティティだけをClient Helloに乗せてサーバ側に送ります。

サーバ側では、クライアントからのPSK接続要求があるとSSL_accept関数の中で、受け取ったアイデンティティをアーギュメントに指定してあらかじめ登録されたコールバック関数を呼び出します。コールバック関数ではアーギュメントで指定されたアイデンティティに対応する鍵値をアーギュメントに返却します。ライブラリーではこの値を使用してPSK接続します。

PSKのモードにはこの鍵値を直接使用するモードと鍵値をもとに(EC)DHEによる鍵共有を行ないセッション鍵を得るモードの二つがあります。通常は後者により鍵共有を行いコールバックで指定した鍵値は直接には使用しません。その場合も(EC)DHEのパラメータと公開鍵がClient HelloとServer HelloのKey Share拡張に示されるので、追加のメッセージを送ることなくServer Helloの直後がら暗号化を開始します。



6.3.2 コマンド形式と使用例

サーバ、クライアントのコマンド形式は「6.1 クライアント・サーバー通信」と同じです。

ここでは、ローカルホストのサーバ(Server-tls-psk)との単純なPSK接続を試してみます。まず、サーバウィンドウでPSKサーバを起動します。

```
$ ./Server-tls-psk
Waiting for a connection...
```

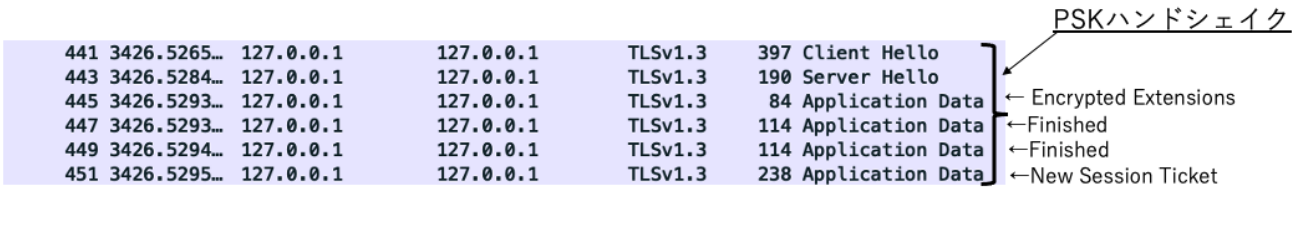
次にクライアントウィンドウでPSKクライアントを起動すると、サーバとPSK接続しクライアント側のアイデンティティ名("Client_identity")を サーバ側に送ります。PSK接続が成立して送信メッセージのプロンプトが出力されます。

```
$ ./Client-tls-psk
Send to localhost(127.0.0.1)
Message to send:
```

この時点でサーバ側には送られてきたアイデンティティ名("Client_idenitity")とPSK接続成立のメッセージが表示されます。

```
Identity: Client_identity
Client connected successfully
```

この時点まででハンドシェークメッセージは下のようになり、6.1の場合と比較するとメッセージに証明書が含まれていないことがわかります。



また、Client Helloにはpsk_key_exchange_modes拡張が含まれていて、クライアント側からPSK接続を要求していることがわかります。 その中には、PSK-onlyとPSK with (EC)DHEの二つの選択肢が示されています。

```
Extension: psk_key_exchange_modes (len=3)
    Type: psk_key_exchange_modes (45)
    Length: 3
    PSK Key Exchange Modes Length: 2
    PSK Key Exchange Mode: PSK-only key establishment (psk_ke) (0)
```

```
    PSK Key Exchange Mode: PSK with (EC)DHE key establishment (psk_dhe_ke)
(1)
```

アイデンティティ名はPre-Shared Key extension拡張に含まれています。"436c69656e745f6964656e74697479" はコールバックで 指定した"Client_identity"であることが確認できます。

```
Pre-Shared Key extension
    Identities Length: 21
    PSK Identity (length: 15)
        Identity Length: 15
        Identity: 436c69656e745f6964656e74697479
        Obfuscated Ticket Age: 0
    PSK Binders length: 33
    PSK Binders
```

一方、Server Helloにはpre_shared_key 拡張が存在し、key_share にECDH secp256r1が合意されたことを示しています。

```
Handshake Protocol: Server Hello
    ...

    Extension: pre_shared_key (len=2)
        Type: pre_shared_key (41)
        Length: 2
        Pre-Shared Key extension
            Selected Identity: 0
    Extension: key_share (len=69)
        Type: key_share (51)
        Length: 69
        Key Share extension
            Key Share Entry: Group: secp256r1, Key Exchange length: 65
                Group: secp256r1 (23)
                Key Exchange Length: 65
                Key Exchange:
04631bfc0c693fce42e72a19beedf44bd8d9fcae6f1813f391fb7d591c29405f8563a876…
    ...
```

6.3.3 プログラム

1) クライアント

```
/* PSKクライアントコールバック */
static inline unsigned int my_psk_client_cb(SSL* ssl, const char* hint,
```

```
        char* identity, unsigned int id_max_len, unsigned char* key,
        unsigned int key_max_len)
{
    /* アイデンティティをアーギュメントにコピー */
    strncpy(identity, kIdentityStr, id_max_len);
    /* 事前共有鍵をアーギュメントにセット */
    key = (unsigned char*)"\x1a\x2b\x3c\x4d";

    /* 鍵長を返却 */
        if (strlen((const char*)key) < key_max_len) {
        return strlen((const char*)key);
    }
    else {
        return 0;
    }
}


int main(int argc, char **argv)
{
    ...
    前処理（6.1 クライアントサンプルと同様）
    ...

    /*  SSLコンテクストの確保し、CA証明書をロード  */
        if(ctx = SSL_CTX_new(SSLv23_client_method()) == NULL)
    { エラー処理  }

    /* PSKコールバックの登録 */
    SSL_CTX_set_psk_client_callback(ctx, my_psk_client_cb);

    以下、6.1 クライアントサンプルと同様
     ...

cleanup:
    リソースの解放
}
```

2) サーバー

```


/* PSKサーバーコールバック */
static unsigned int my_psk_server_cb(SSL* ssl, const char* identity,
                            unsigned char* key, unsigned int key_max_len)
{

    /* 受け取ったidentityが期待するものかチェック */
    printf("Identity: %s\n", identity);
        if (strncmp(identity, "Client_identity", 15) != 0) {
        printf("error!\n");
        return 0;
```

```
    }

    /* 事前共有鍵をアーギュメントにセット */
    key = (unsigned char*)"\x1a\x2b\x3c\x4d";

    /* 鍵長を返却 */
        if (strlen((const char*)key) < key_max_len) {
        return strlen((const char*)key);
    }
    else {
        return 0;
    }
}


int main(int argc, char **argv)
{
    ...
    前処理（6.1 サーバに同じ）
    ...

    /*  SSLコンテクストの確保し、サーバー証明書、プライベート鍵をロード  */
        if ((ctx = SSL_CTX_new(SSLv23_server_method())) == NULL)
    { エラー処理  }

    /* PSKコールバックの登録 */
    SSL_CTX_set_psk_server_callback(ctx, my_psk_server_cb);

    while (1) {

        ...
        TCP、TLSアクセプト（6.1サーバに同じ）
        ...

        while(1) {

            ...
            アプリケーションメッセージの送受信（6.1サーバに同じ）
            ...

        }
    }

cleanup:
    リソースの解放（6.1サーバに同じ）
}
```

主な管理構造体とAPI

- コールバック関数の登録

  クライアント側 : SSL_CTX_set_psk_client_callback

> 第2引数でクライアント側のコールバック関数を登録します。コールバック関数はSSL_connect呼び出し時に、事前鍵およびIDの取得のために呼び出されます。事前鍵はクライアント側に保持され、IDのみサーバー側に送られます。

サーバー側: SSL_CTX_set_psk_server_callback

> 第2引数でサーバー側のコールバック関数を登録します。コールバック関数はSSL_accept呼び出し時処理中に呼び出されます。引数にクライアントから受け取ったIDが渡されるので、コールバック処理はIDに対応する事前鍵を返却します。

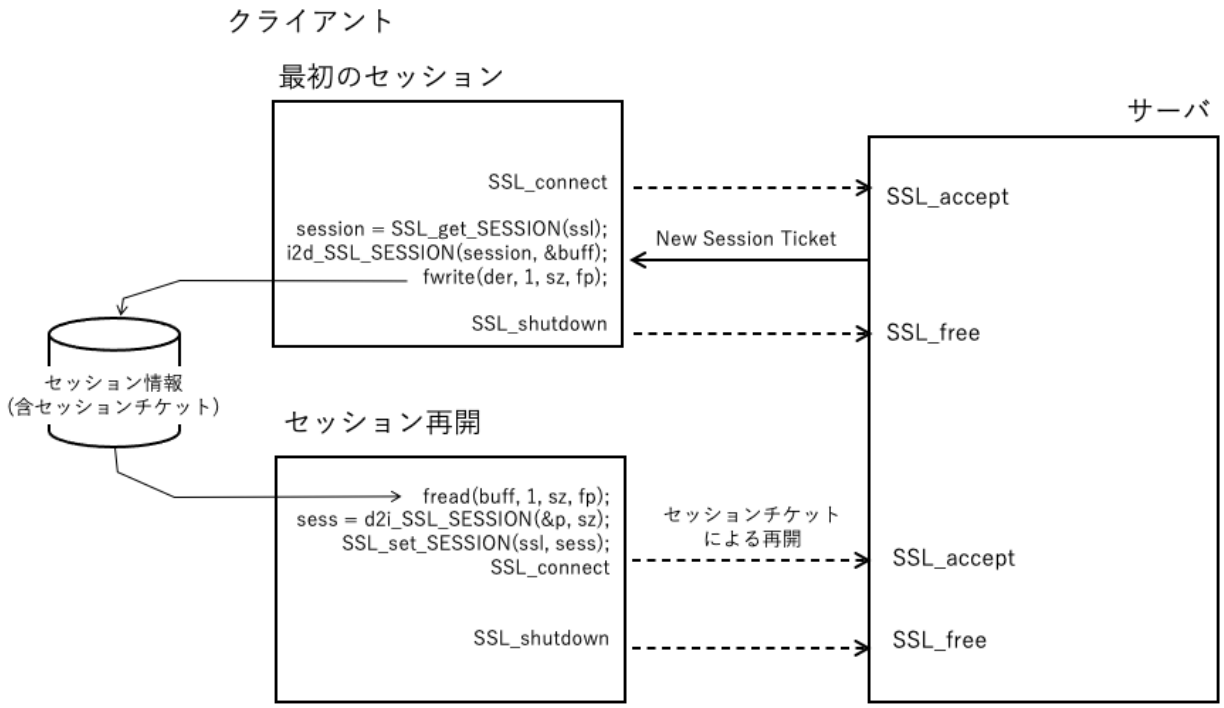
6.4 セッション再開

6.4.1 概要

このサンプルでは、セッション再開によるメッセージ通信を行います。

クライアントプログラムとして、最初のセッション用(Client-tls-session) とセッション再開用(Client-tls-resume) の二つを使用します。 サーバは「6.1 クライアントサーバ間通信」のサーバ(Server-tls)をそのまま利用します。

最初のセッションではクライアントはサーバーからセッションチケットを受け取りファイルに保存しておきます。 セッション再開のクライアントでは、ファイルに保存したセッション情報を読み出し、それを使用して再開します。



最初のセッションでクライアントとサーバのハンドシェークが完了し安全なセッションが確立すると、サーバから New Session Ticketメッセージが 送信されます。クライアントは送られたチケットをファイルに格納しておきます。

セッション再開のクライアントは起動時にファイルからチケットを読み出しコンテクストに登録しておきます。 SSL_connect時には登録されたセッションチケットが サーバに送られサーバ側ではそのチケットを使用してセッションを再開します。

6.4.2 コマンド形式と使用例

コマンドアーギュメントの形式は「6.1 サーバクライアント間通信」のサーバ、クライアントと同じです。

最初にサーバウィンドウでサーバ側を起動しておきます。

```
$ ./Server-tls
Waiting for a connection...
```

次に最初のセッション用のクライアントを起動し、フルハンドシェイクによるセッションを確立します。この時、サーバからNew Session Ticketが 送られてきているはずなので、"break" にてサーバへの接続を終了します。この時、セッション情報はファイル(session.bin)に保存します。

```
$ ./Client-session-tls
Send to localhost(127.0.0.1)
Message to send: Hello server
Received: I hear ya fa shizzle!
Message to send: break
End of TLS Client
```

サーバ側でも、最初のセッションに対応するメッセージが出力されます。最初のセッション終了後、次の接続待ち状態に入ります。

```
Client connected successfully
Received: Hello server
Received: break
Received break command
Closed the connection
Waiting for a connection...
```

ここまでで、Wireshark上でも暗号化されているもののフルハンドシェイクとアプリケーションメッセージのやりとりが確認できます。

No.	Time	Source	Destination	Protocol	Length	Info
5	0.001716	127.0.0.1	127.0.0.1	TLSv1.3	312	Client Hello
7	0.003624	127.0.0.1	127.0.0.1	TLSv1.3	184	Server Hello
9	0.004412	127.0.0.1	127.0.0.1	TLSv1.3	84	Application Data
11	0.004446	127.0.0.1	127.0.0.1	TLSv1.3	1020	Application Data
13	0.008345	127.0.0.1	127.0.0.1	TLSv1.3	342	Application Data
15	0.008371	127.0.0.1	127.0.0.1	TLSv1.3	114	Application Data
17	0.008506	127.0.0.1	127.0.0.1	TLSv1.3	114	Application Data
19	0.008620	127.0.0.1	127.0.0.1	TLSv1.3	238	Application Data
21	7.262450	127.0.0.1	127.0.0.1	TLSv1.3	90	Application Data
23	7.262562	127.0.0.1	127.0.0.1	TLSv1.3	100	Application Data
25	9.829919	127.0.0.1	127.0.0.1	TLSv1.3	83	Application Data
27	9.830322	127.0.0.1	127.0.0.1	TLSv1.3	80	Application Data
31	9.831596	127.0.0.1	127.0.0.1	TLSv1.3	80	Application Data

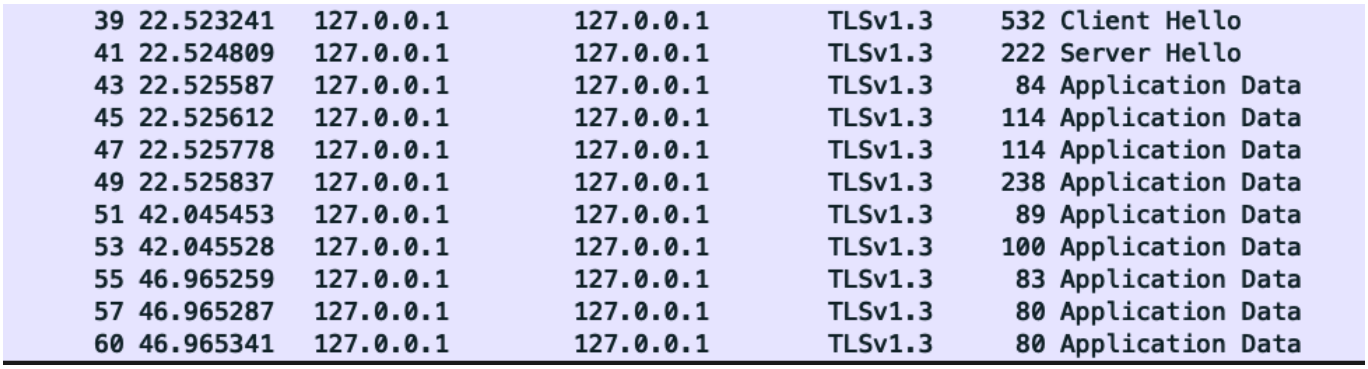
次に、セッション再開用のクライアントを起動します。このクライアントは先ほどファイルに保存したセッション情報を読み出し、セッション再開を実行します。 セッションが再開されたら送信メッセージの入力プロンプトが表示されるので適当なメッセージを入力し、"break"でセッションを終了します。

```
$ ./Client-resume-tls
Send to localhost(127.0.0.1)
session.bin size = 255
Resuming session
Session is reused
Message to send: Hello again
Received: I hear ya fa shizzle!
Message to send: break
Sending break command
End of TLS Client
```

その間、サーバ側でも受信したメッセージが表示されます。

```
Client connected successfully
Received: Hello again
Received: break
Received break command
Closed the connection
Waiting for a connection...
```

その間、Wireshark上にはセッション再開分のメッセージが同様に追加表示されます。



39	22.523241	127.0.0.1	127.0.0.1	TLSv1.3	532 Client Hello
41	22.524809	127.0.0.1	127.0.0.1	TLSv1.3	222 Server Hello
43	22.525587	127.0.0.1	127.0.0.1	TLSv1.3	84 Application Data
45	22.525612	127.0.0.1	127.0.0.1	TLSv1.3	114 Application Data
47	22.525778	127.0.0.1	127.0.0.1	TLSv1.3	114 Application Data
49	22.525837	127.0.0.1	127.0.0.1	TLSv1.3	238 Application Data
51	42.045453	127.0.0.1	127.0.0.1	TLSv1.3	89 Application Data
53	42.045528	127.0.0.1	127.0.0.1	TLSv1.3	100 Application Data
55	46.965259	127.0.0.1	127.0.0.1	TLSv1.3	83 Application Data
57	46.965287	127.0.0.1	127.0.0.1	TLSv1.3	80 Application Data
60	46.965341	127.0.0.1	127.0.0.1	TLSv1.3	80 Application Data

しかし、Client HelloとServer Helloの詳細を見ると、プロトコル上はPSKと同様の扱いでセッション再開が実現されていることがわかります。

Client Hello:

```
TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    ...

    Extension: psk_key_exchange_modes (len=3)
        Type: psk_key_exchange_modes (45)
        Length: 3
        PSK Key Exchange Modes Length: 2
```

```
        PSK Key Exchange Mode: PSK-only key establishment (psk_ke) (0)
        PSK Key Exchange Mode: PSK with (EC)DHE key establishment
(psk_dhe_ke) (1)
    Extension: key_share (len=71)
        Type: key_share (51)
        Length: 71
        Key Share extension
            Client Key Share Length: 69
            Key Share Entry: Group: secp256r1, Key Exchange length: 65
                Group: secp256r1 (23)
                Key Exchange Length: 65
                Key Exchange:
0486fb0b50d08da72595027a3a7cc307f075008239e7cdbe9a7db50523aceb9eba5ac8a1…
    ...

    Extension: pre_shared_key (len=185)
        Type: pre_shared_key (41)
        Length: 185
        Pre-Shared Key extension
            Identities Length: 148
            PSK Identity (length: 142)
                Identity Length: 142
                Identity:
a574646e6cdf354513f74092c47a9074e8e4c7342942a70f3a171cf25c868013004c29b8…
                Obfuscated Ticket Age: 3369942077
            PSK Binders length: 33
            PSK Binders]
```

Server Hello:

```
TLSv1.3 Record Layer: Handshake Protocol: Server Hello
    ...

    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    ...
    Extension: pre_shared_key (len=2)
        Type: pre_shared_key (41)
        Length: 2
        Pre-Shared Key extension
            Selected Identity: 0
    Extension: key_share (len=69)
        Type: key_share (51)
        Length: 69
        Key Share extension
            Key Share Entry: Group: secp256r1, Key Exchange length: 65
                Group: secp256r1 (23)
                Key Exchange Length: 65
                Key Exchange:
04d9f17a76094cceefc17f43acc3ea255b909bb348b6204dfecdba4935953e5d8d688af0…
```

6.4.3 プログラム

1) クライアント：最初のセッション

```
/* セッションを保存 +/
static int write_SESS(SSL *ssl, const char* file)
{
        if ((fp = fopen(file, "wb")) == NULL)
    { エラー処理 }

    /* セッション情報をDER形式に変換 */
        if ((sz = i2d_SSL_SESSION(sess, &buff)) <= 0)
    { エラー処理 }

        if ((fwrite(buff, 1, sz, fp)) != sz)
    { エラー処理 }

cleanup:
    リソース解放
    ...

    return ret;
}

int main(int argc, char **argv)
{

    ...
    前処理（6.1 クライアントに同じ）
    ...

    while(1) {

        ...
        メッセージ入力、サーバに送信（6.1 クライアントに同じ）
        ...

        /* メッセージが "break" の場合、セッションを保存して終了 */
            if (strcmp(msg, "break") == 0) {
            session = SSL_get_session(ssl);
            ret = write_SESS(session, SAVED_SESS);
            break;
        }

        ...
        メッセージ受信、表示（6.1 クライアントに同じ）
        ...
    }

cleanup:
```

```
    リソースの解放
}
```

2) クライアント：セッション再開

```
/* セッション読み出し  */
static int read_SESS(const char* file, SSL* ssl)
{
    ...
    ファイルオープン、ファイルサイズ(sz)を取得
    ...

    /* バッファを確保し、セッション情報読み込み */
        if ((buff = (unsigned char*)malloc(sz)) == NULL ||
        (fread(buff, 1, sz, fp) != sz))
        エラー処理
    }

    printf("%s size = %ld\n", SAVED_SESS, sz);

    /* セッション情報をDERから内部形式に変換 */
    p = buff;
        if((sess = d2i_SSL_SESSION(NULL, (const unsigned char**)&p, sz))
== NULL)
    { エラー処理 }

    /* セッション情報を設定 */
        if(sess != NULL && (ret = SSL_set_session(ssl, sess) !=
SSL_SUCCESS)) {
        print_SSL_error("failed SSL session", ssl);
    } else {
       printf("Resuming session\n");
       ret = SSL_SUCCESS;
    }

cleanup:
    リソース解放
}

int main(int argc, char **argv)
{
    ...
    前処理（6.1 クライアントに同じ）
    ...

    /*  SSLオブジェクトの確保  */
        if ((ssl = SSL_new(ctx)) == NULL)
    { エラー処理 }

    /* ファイルからセッション情報を読み出し、設定 */
```

```
        if ((ret = read_SESS(SAVED_SESS, ssl)) != SSL_SUCCESS)
    { エラー処理 }

    以下、6.1 クライアントに同じ

cleanup:
    リソースの解放
}
```

6.4.4 主な管理構造体とAPI

- SSL_SESSION
    - SSL構造体で管理されている接続情報のうち、セッションチケットなどセッション再開で必要とされる情報一式を抽出し管理する構造体です。
- 構造体の確保、設定と解放
    - 確保：SSL_get_SESSION()
      SSL構造体の接続情報からセッション再開に必要なデーター式をSSL_SESSION構造体の形で抽出します。そのときSSL_SESSION構造体に必要な領域を確保し、そのポインタを返却します。
      SSL_get_SESSIONはクライアントがSSL_connectを実行後、TLSの安全な接続が確保されている間に呼び出します。
    - 設定：SSL_set_SESSION()
      SSL_get_SESSIONで取り出したSSL_SESSION構造体をセッション再開のためにSSL構造体に設定します。SSL_set_SESSIONはクライアントでSSL_connectを行う前に呼び出します。
    - 解放：SSL_SESSION_free()
      SSL_SESSION構造体を解放します。
- 構造体のデータ変換
    - ANS1形式から内部形式へ : d2i_SSL_SESSION
      d2i_SSL_SESSIONは、ASN1形式で保存されたSSL_SESSION構造体のデータを内部形式のSSL_SESSION構造体へ再構築し、そのポインタを返却します。
    - 内部形式からASN1形式へ : i2d_SSL_SESSION
      i2d_SSL_SESSIONは、内部形式のSSL_SESSION構造体データをASN1形式のデータへ変換します。変換データは第2引数で渡されたポインタに必要なメモリを確保し設定します。関数の戻り値はASN1形式変換に必要な長さを返却します。第2引数にNULLが渡された場合でも、関数の戻り値はASN1形式変換に必要な長さを返却します。

6.5 Early Data (0-RTT)

6.5.1 概要

このサンプルでは、6.4 セッション再開のサンプルプログラムをベースにEarly Data (0-RTT) によるメッセージ通信を行います。

Early Data使用上の注意

Early Dataは前のセッションの鍵を利用しているので前方秘匿性が低いためクリティカルな情報の送受信に使用してはいけません。また、リプレイ攻撃に弱い点もあるので、 実用のプログラムでは何らかの形であらかじめ送受信者がデータの内容についてある程度の合意に基づいてアプリケーション側リプレイ攻撃を判別できるようにしておく配慮も必要です。 ここでのサンプルプログラムでは基本機能を理解するために基本機能のみで、そのような配慮はされていない点に注意してください。

6.5.2 コマンド形式と使用例

コマンドの実行手順は「6.4 セッション再開」と同じです。

サーバ側のウィンドウでサーバ(Server-tls-eld)を起動し、クライアント側で最初のセッション(Client-tls-session)を実行します。

```
$ ./Server-tls-eld
Waiting for a connection...
Client connected successfully
Received: Hello server
Received: break
Received break command
Closed the connection
Waiting for a connection...
```

```
$ ./Client-tls-session
Send to localhost(127.0.0.1)
Message to send: Hello server
Received: I hear ya fa shizzle!
Message to send: break
End of TLS Client
```

最初のセッションを終了すると、サーバ側では次のセッションの待ち状態になるので、Early Data付きのセッション再開クライアント(Client-tls-eld)を 起動します。セッション再開時にEarly Dataが送信され、アプリケーションメッセージ入力のプロンプトが表示されます。メッセージを入力して セッションを終了します。

```
$ ./Client-tls-eld
Send to localhost(127.0.0.1)
session.bin size = 263
Resuming session
Early Data: good early morning
Session is reused
Early Data was accepted
Message to send: Hello again
Received: I hear ya fa shizzle!
Message to send: break
Sending break command
End of TLS Client
```

サーバ側では、セッション再開で受信したEarly Dataとアプリケーションメッセージが表示されます。

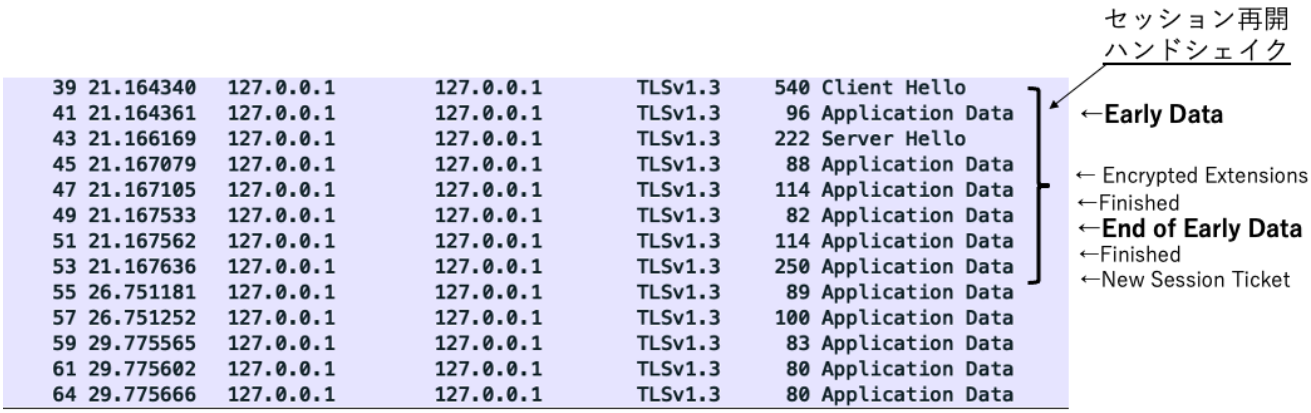
```
Early Data Client message: good early morning
Client connected successfully
Received: Hello again
Received: break
Received break command
Closed the connection
Waiting for a connection...
```

この間、Wiresharkでパケットをモニターしていると、後半のセッション再開部分は下のようなパケットがキャプチャーできます。

Client Helloにはearly_data拡張が含まれていて、このセッションにはEarly Dataが存在することを示しています。

```
Extension: early_data (len=0)
    Type: early_data (42)
    Length: 0
```

Client Helloの後に通常のApplication Dataレコードと同じ形式で暗号化されたEarly Dataが送信されているのがわかります。



また、「6.4 セッション再開」で見た通常のセッション再開のパケットと比べると、クライアントからのFinished の前にEarly Dataの 終了を示す "End of Early Data" メッセージも追加されています。

6.5.3 プログラム

1. クライアント：最初のセッション

クライアント側の最初のセッションです。TLS接続後、適当なタイミングでsセッションチケットを含むセッション情報を保存します。このプログラムは、6.1 のクライアントをそのまま利用します。

2. クライアント：セッション再開

このクライアントプログラムではサーバとのセッションを再開し、再開時にEarly Data(0-RTT) を送信します。Early Dataに関する部分以外のセッション再開の流れは6.4セッション再開のセッション再開用クライアントと同じです。

```
/* Ealy Data(0-RTT)のステータス取得  */
static void EarlyDataStatus(SSL* ssl)
{
    if (earlyData_status < 0) return;

    printf("Early Data is ");

    switch(earlyData_status) {
        case SSL_EARLY_DATA_NOT_SENT:
                printf("not sent.\n");
                break;
        case SSL_EARLY_DATA_REJECTED:
                printf("rejected.\n");
                break;
        case SSL_EARLY_DATA_ACCEPTED:
                printf("accepted\n");
                break;
        default:
                printf("unknown...\n");
    }
}

/* Ealy Data(0-RTT)の送信  */
static int writeEarlyData(SSL* ssl, const char* msg, size_t msgSz)
{
    /* Ealry Data の書き出し */
    ret = SSL_write_early_data(ssl, msg, msgSz, &writtenbytes);
        if (msgSz != writtenbytes || ret <= 0)
        エラー処理
    else
        ret = SSL_SUCCESS;
    return ret;
}

int main(int argc, char **argv)
{
    ...
    前処理（6.4 セッション再開用クライアントに同じ）
    ...

    if(writeEarlyData(ssl, kEarlyMsg, sizeof(kEarlyMsg)-1)) !=
SSL_SUCCESS)
```

```
    {    エラー処理  }

    if((ret = SSL_connect(ssl)) != SSL_SUCCESS) {
     {    エラー処理  }

    /* Early Data の状態確認  */
    EarlyDataStatus(ssl);


    /* アプリケーションメッセージ送受信 */
    while(1) {
        ...
        6.4 セッション再開用クライアントに同じ
        ...
    }

cleanup:
    リソースの解放
}
```

3. サーバー側での Early Data \受信

```
static void ReadEarlyData(SSL* ssl)
{
    do {
         ....
         /* Early Data の読み出し*/
         ret = SSL_read_early_data(ssl, early_data, sizeof(early_data)-1,
&len);
            if (ret <= 0) {
             エラー処理
         }
         /* 受信したEarly Dataを表示 */
        if (len > 0) {
            early_data[len] = '\0';
            printf("Early Data Client message: %s\n", early_data);
        }
      } while(ret > 0);

}
int main(int argc, char **argv)
{


    ...
    前処理（6.5 セッション再開のサーバに同じ）
    ...

    while (1) {

        ...
        TCPアクセプト（6.5セッション再開のサーバに同じ）
```

```
        ...

        /* Early Data の受信 */
        if ((ret = SSL_set_max_early_data(ssl, MAX_EARLYDATA_SZ)) !=
SSL_SUCCESS)
        { エラー処理 }

        /* Early Dataの受信 */
        ReadEarlyData(ssl);

        /* TLSアクセプト */
        if ((ret = SSL_accept(ssl)) != SSL_SUCCESS)
        { エラー処理 }

        while(1) {

            ...
            アプリケーションメッセージの送受信（6.5セッション再開のサーバに同じ）
            ...

        }
    }

cleanup:
    リソースの解放（6.5セッション再開のサーバに同じ）
}
```

6.5.4 プログラムの説明：

Early Data処理部分を除いて、処理は6.5 セッション再開と同じです。

- クライアント側セッション再開時のEarly Dataの送信: SSL_write_early_data

  TLSのセッション再開時にEarly Dataを送信します。送信はセッション再開ハンドシェークの一環として行われ、Early Dataの 送信が完了した時点で関数はリターンします。その後、SSL_connectを呼び出しハンドシェークの残り部分を完了させます。

- サーバ側セッション再開時のEarly Dataの受信 : SSL_read_early_data
  TLSの安全なセッションが再開され、クライアントからEarly Dataが送信されている場合にEarly Data を受信し、関数からリターンします。 その後。SSL_accesptを呼び出してハンドシェークの残り部分を完了させます。