MWA Software

IndySec OpenSSL User Guide

COPYRIGHT

The copyright in this work is vested in MWA Software. The contents of the document may be freely distributed and copied provided the source is correctly identified as this document.

Disclaimer

Although our best efforts have been made to ensure that the information contained within is up-to-date and accurate, no warranty whatsover is offered as to its correctness and readers are responsible for ensuring through testing or any other appropriate procedures that the information provided is correct and appropriate for the purpose for which it is used.

CONTENTS   Page

1

Introduction

This User Guide is intended to provide an introduction to the IndySecOpenSSL package. It covers the background and history of the package, its purpose, how to install and how to use it. Appendices also provide supporting material on the RFC 8446 Transport Layer Security (TLS) protocol, and related security concepts including X.509 certificates.

1.1   Background

Indy is a well-known internet component suite for **Delphi**, **C++Builder**, and **Free Pascal** providing both low-level support (TCP, UDP, raw sockets) and over a 120 higher level protocols (SMTP, POP3, NNT, HTTP, FTP) for building both client and server applications.

The package also supports the HTTPS protocol using the OpenSSL library to provide secure communication using the Transport Level Security (TLS) protocol.

However, at the time of writing, the current release of Indy (10.6.x), only supports the obsolete OpenSSL 1.0.2 release. This release of OpenSSL no longer receives updates from the OpenSSL project and only supports the the TLS 1.2 protocol. This is a serious problem for Indy users that need to use an up-to-date release of OpenSSL (3.x) which is both supported and supports the current TLS 1.3 protocol.

In mid-2024, the Indy Maintainer asked others for help to add OpenSSL 3.x support to the forthcoming 10.7 release, and to create an OpenSSL support package separate from the main Indy release.

A strawman package was produced. There was no immediate sight of Indy 10.7 and the contributors then went down different paths using the same initial package. One direction resulted in the release of an add-on to Indy 10.6 which required changing the unit and class names (TaurusTLS).

A common basis for this work are the Pascal OpenSSL API bindings - see

https://github.com/MWASoftware/PascalAPI4OpenSSL

These are generally available for Pascal developers irrespective of whether or not they use the Indy library.

The originator of these bindings also developed a fork of Indy 10.6 with OpenSSL 3.x support. This is the Indy.ProposedUpdate -see

https://github.com/MWASoftware/Indy.proposedUpdate

This repo has been kept this up-to-date with Indy 10.6 commits.

The same OpenSSL units were also copied to a fork of the Indy Project's proposed IndyTLS-OpenSSL package -see

https://github.com/MWASoftware/IndyTLS-OpenSSL

for use with the putative Indy 10.7. At the very least, Indy users had a choice of upgrade path.

In terms of the code, the Indy.ProposedUpdate and the TaurusTLS software has diverged since then. The Indy.ProposedUpdate code base, since the initial strawman, has improved the OpenSSL library load strategies with the faster "Just in Time" approach. There has also been considerable code clean up in the meantime in respect of the Indy OpenSSL code, and emphasis has been put on ensuring that both Delphi and Lazarus are fully supported by the same codebase - plus many bug fixes

Indy 10.7 still seems to be as far off as it was in 2024. It is no longer feasible to keep maintaining the Indy.ProposedUpdate and the IndyTLS-OpenSSL codebases in the hope that Indy 10.7 will some day be released. A clear direction needs to be provided for Indy users.

These codebases have now been replaced with an independent package IndySecOpenSSL. This package is a clone of the IndyTLS-OpenSSL codebase, but applies the TaurusTLS strategy of changing the class and unit names so that it can be used as an add on to Indy 10.6, and will also be suitable for use with Indy 10.7. Existing users can readily upgrade by simply changing the SSL class names to their updated equivalents and references to the IdSSLOpenSSL unit with a reference to the replacement IdSecOpenSSL unit (see 3.1 for detailed guidance).

The name has been chosen to reflect its history - much of the code was originally developed by Indy Developers - and the intent to provide OpenSSL support to Indy. The same Open Source licences are also used, A further intent is that this package becomes the provider of OpenSSL support to Indy.

1.2   Purpose and Objectives

This package provides a new (optional) OpenSSL package separate from Indy's "protocols" package and adds support for OpenSSL 3.0 and later. An important objective is that both Delphi and Lazarus/fpc are fully supported by this source code tree.

It's purpose is to provide Indy users with an upgrade path to the use of current OpenSSL libraries with the minimum of change. This includes users that use the existing version of Indy bundled with Delphi and the version provided with the Lazarus Online Package Manager.

In terms of the deployment of user applications that use the IndySecOpenSSL package, three link models are supported.

* Dynamic Library Load (the default and the approach used in previous versions)

* compile time linkage to a shared (.so or .dll) library (OpenSSL 3.x only)

* compile time linkage to a static library (FPC only with gcc compiled OpenSSL).

The package is primarily intended for use with OpenSSL 3.x. However, backwards compatibility is also provided so that applications that must continue to use OpenSSL 1.0.2 or 1.1.1 may continue to do so (see 3.4).

1.3   Conventions Used in this User Guide

1. In path names, the symbol <indy> expands to the installation folder for the Indy source code.

2. In path names, the symbol <indysec> expands to the installation folder for the IndySecOpenSSL source code.

2

Installation of IndySecOpenSSL

2.1   Downloading the Software

The IndySecOpenSSL package is currently only available from github. For Lazarus users, there is also the intention that it should soon be available using the Lazarus Online Package Manager (OPM).

The package can be downloaded from github either as a zip or tar.gz archive or cloned using the *git* utility.

2.1.1   Download of a zip or tar.gz Archive

The latest stable release of IndySecOpenSSL can be downloaded by pointing your browser at https://github.com/MWASoftware/IndySecOpenSSL and clicking on the "latest Release" button at the right hand side of the page (Desktop versions). The layout of smaller screen varies, but the "Releases" section should always be present when scrolling down Once you have selected the latest release, the download page should open allowing you to select the zip or tar.gz archive depending on your preference.

The downloaded archive should then be expanded into some suitable and permanent location on your local computer.

2.1.2   Cloning Using git

git is a widely available command line utility available for both Linux and Windows.

Linux:	*git* is normally installed from your Linux distros repository. For example, on Ubuntu, Linux Mint or other Debian derived distributions, entering the following at the command line: `sudo apt-get install git`

	is usually sufficient to install git.
Windows:	You can download the latest installation package from https://git-scm.com/downloads/win

Once git has been installed. The IndySecOpenSSL package may be downloaded from the command line using:

```
git clone https://github.com/MWASoftware/IndySecOpenSSL.git
```

> *Note: when run under windows this command line assumes that the installed location of git has been added to your PATH environment variable by the installer. If not, then you may need to use the full path to git, typically "C:\Program Files\Git\bin\git.exe".*

This command will download the current up-to-date source code tree into the sub-directory IndySecOpenSSL.

Your local copy may, at any time, be brought up-to-date by entering, when IndySecOpenSSL is your current directory, at the command line:

```
git pull origin main
```

> *Note: the above command assumes that "origin" remote repository has been created by git as pointing to the github repo and that you are working on the main branch. i.e. that you have not changed any defaults.*

2.2 The OpenSSL Library Files

The OpenSSL library files are separate to the IndySecOpenSSL package but are essential for development and are also required for most deployments.

- Development systems must have all OpenSSL library files available. This includes static libraries if you wish to deploy applications with OpenSSL linked into the executable. However, 'C' header files, documentation and examples are not required.

- Deployed applications, other than statically linked (to OpenSSL) applications, must have access to the runtime OpenSSL DLLs (Windows) or shared objects (Posix systems).

A root certificate store should also be available for both testing and deployment.

> *Note. OpenSSL trusts and uses the root certificate store when authenticating remote clients. Any corruption by accident or malware can result in failed authentication or the incorrect authentication of an attacker's website. It is important that you ensure that the root certificate store is appropriately protected from malware and any other unauthorised access.*

2.2.1 Linux

Most Linux distributions install OpenSSL by default. This includes shared libraries and root certificates. In most cases no further action needed by taken. The IndySecOpenSSL package when used with shared libraries will automatically locate and load the installed OpenSSL Libraries. This is true for both development and target systems.

If you do need to explicitly install OpenSSL then, for example, on Debian and derivatives the *libssl3t64* package provides the shared OpenSSL 3.x libraries and the *ca-certificates* package provides the root certificates.

However, the static libraries are not installed by default. If required, they are provided (Debian) by the *libssl-dev* package and this must be explicitly installed..

Other Linux distros may vary the package names, but the result is similar.

You may then rely on your distro for updates and bug fixes to the OpenSSL libraries and the root certificate store.

2.2.2   Windows

OpenSSL libraries are not included in Windows itself and there is no "official" distribution for Windows. The OpenSSL Wiki (https://github.com/openssl/openssl/wiki/Binaries) lists several sources of OpenSSL libraries compiled for Windows, but without endorsing any listed provider. It is up to the user to select a source and to do their own due diligence as the trustworthiness of the source.

Alternatively, you can always compile your own version...

In all cases, both the libcrypto and libssl dlls are needed. For example, for OpenSSL 3.x and Windows 64-bit these libraries are typically named:

- libcrypto-3-x64.dll, and
- libssl-3-x64.dll

32-bit versions usually omit the "-x64" suffix.

For reliable results, these DLLs should be located in the same folder as your application's executable (both testing and deployment). It is possible to specify an alternative location, but this is the simplest approach, and the one that should always result in the OpenSSL library that you have selected being used.

The location of the OpenSSL libraries under Windows is further discussed in 3.3.

2.2.2.1   The Windows Root Certificate Store

OpenSSL libraries do not usually come with a set of root certificates and, if they do, it is wise not to trust their provenance. On the other hand, Windows does provide its own root certificate store which is maintained by Windows Update. This is recommended for most users.

> *Note: there is always the usual caveat when it comes to trusting anything to so with Windows. That is there is always a risk that malware may corrupt the Windows Root Certificate store and its use is always dependent on suitable protections being in place.*

By default, IndySecOpenSSL on Windows will load the Windows Root Certificate store and make it available to OpenSSL as an in memory certificate store instead of any root certificates that may be located in the compiled in location for root certificates.

Advanced users that wish to control and mange the root certificate store themselves can override this default (see 4.3.1).

2.3   Installation into Delphi

Indy 10.6 or 10.7 must be installed into Delphi before IndySecOpenSSL is installed. Currently Indy 10.6 is bundled with Delphi and installed by the Delphi installer. To otherwise install Indy or to upgrade from the bundled version, please consult the Indy documentation.

> *Note. You do not have to install dclIndySecOpenSSL nto the IDE in order to run the console mode examples (see 3.5).*

2.3.1   IndySecOpenSSL installation

Download the IndySecOpenSSL package as described above.

IndySecOpenSSL may then be installed any time after Indy has been installed into the Delphi IDE

To install IndySecOpenSSL, in the Delphi IDE, open the <indysec>\delphi\IndySec.grouproj project group and right click on the dclIndySecOpenSSL project in the Delphi IDE Project Group Window. Select "install".

IndySecOpenSSL should now be installed into Delphi. The IndySecOpenSSL components should then be available on the "IndySec" pallet tab.

> *Note: when Indy 10.6 is installed, the obsolete Indy 10.6 OpenSSL components are still installed and visible on the "I/O Handlers - Protocols" pallet tab. Unfortunately, they cannot be readily uninstalled without modifying the Indy 10.6 packages. However, while they should not interfere with the IndySec components, there is a residual risk that the user may select them by mistake rather than using their replacemetns on the IndySec tab.*

> *Note. The IndySecOpenSSL package is dependent on IndySystem, IndyCore, IndyProtocols. This is correct for use with the Indy 10.6 distributed with Delphi. However, if you replace this with an early version of Indy 10.7 then you may have to change this requirement to IndySystem290, IndyCore290 and IndyProtocols290. This is to match the package names in the 10.7 version on github..*

2.4   Installation into the Lazarus-IDE

Indy 10.6 or 10.7 must be installed into Lazarus before installing IndySecOpenSSL. Indy 10.6 is available from the Lazarus OPM and this is the recommended route for installing Indy into Lazarus. Alternatively, download Indy from github (https://github.com/IndySockets/Indy) and install the <indy>/Lib/indylaz.lpk package into Lazarus.

IndySecOpenSSL installation:

Download the IndySecOpenSSL package as described above.

In the Lazarus IDE, select the "Package->Open Package File" menu item and navigate to and open the <indysec>/lazarus-fpc/indylaz_openssl.lpk" package file. The Package Manager should now open (see below).

Click on the down arrow next to the "Use" button and select install from the drop down menu. Lazarus should now recompile and restart.

IndySecOpenSSL should now be installed into Lazarus. The IndySecOpenSSL components should be available on the "IndySec" pallet tab.

2.4.1   indylaz.lpk Issues

These are known issues with the Indy software itself and not IndySec, but which can nevertheless impact on its use.

2.4.1.1 Failed Rebuild with "checksum" failure

Sometimes the indylaz package can fail to rebuild with a file checksum issue reported. This can sometimes be worked around with a "cleanup and rebuild". However, a more reliable strategy is to add "-Ur" to the indylaz package "Custom Options".

2.4.1.2 Console Mode Problems

A console mode program that uses IndySecOpenSSL should require the IndySecOpenSSL runtime package. However, as noted above this package is dependent on the indylaz package.

indylaz.lpk is a combined design time and runtime package and thus references the LCL. This can cause problems with console mode programs when it is undesirable to include GUI units. To avoid this issue, it may be necessary to not require any Indy packages in your application and, instead, include:

- <indy>/Lib/System
- <indy>/Lib/Core
- <indy>/Lib/Protocols
- <indysec>/src
- <indysec>/src.opensslhdrs

in your project's search paths.

In the longer term, this issue may be solved in Indy 10.7 by splitting up indylaz into separate design time and runtime packages, as is already the case for Delphi. This approach is currently taken by the Indy.ProposedUpdate for of Indy 10.6, where there are separate indyprotocols, indycore and indysystem run time packages.

For the foreseeable future and in order to remain compatible with Indy 10.6 the IndySecOpenSSL runtime package will be shipped with a dependency on indylaz. However, if you are using a version of Indy that includes the separate indyprotocols, indycore and indysystem run time packages, then you may wish to replace IndySecOpenSSL's dependency on indylaz with a dependency on indyprotocols.

3

Using IndySecOpenSSL

IndySecOpenSSL provides two non-visual components:

- `TIdSecIOHandlerSocketOpenSSL`, and
- `TIdSecServerIOHandlerSSLOpenSSL`.

These are Indy IO Handlers and provide secure SSL/TLS communications to Indy protocol components, such as an http client (TIdHTTP) or an http server (TIdHTTPServer).

These components may be used with existing Indy 10.6 deployments and the forthcoming Indy 10.7. Existing Indy 10.6 projects that use the legacy OpenSSL components will need to be upgraded prior to use of the IndySec components.

> *Note. FIPS support in Indy 10.6 is hard linked to the built in OpenSSL support. This will not be available when only OpenSSL 3.x libraries are available.*

3.1 Upgrading Existing Projects that use Indy and OpenSSL

So that IndySec can be installed and used without having to modify an Indy 10.6 installation, such as that currently provided with Delphi, several class and other type names have been changed, along with their unit names. The original `IdSSLOpenSSL` unit has also been split into several units in order to improve readability and maintainability.

For type names, the general rule has been to replace the "IdSSL" prefix previously used with "IdSec". The notable exception is the `TIdServerIOHandlerSSLOpenSSL` class which become `TidSecServerIOHandlerSSLOpenSSL`.

3.1.1   Unit Split

IdSSLOpenSSL has been split into the following units:

IdSecOpenSSL   Provides `TIdSecIOHandlerSocketOpenSSL` and `TIdSecServerIOHandlerSSLOpenSSL` classes and supporting types.

IdSecOpenSSLSocket   Provides `TIdSecSocket, TIdSecContext` and `TIdSecCipher` classes and supporting types.

procedures LoadOpenSSLLibrary and UnLoadOpenSSLLibrary. These are used for Dynamic load only and normally not called by user, given that the OpenSSL library is automatically loaded on first use and unloaded at program end).

IdSecOpenSSLOptions   Provides `TIdSecOptions` and supporting types.

IdSecOpenSSLX509   Provides `TIdX509, TIdX509SigInfo, TIdX509Fingerprints, TIdX509Info` and `TIdX509Name` classes and supporting types.

IdSecOpenSSLUtils   miscellaneous support functions

IdSecwincrypt   contains a (limited) interface to wincrypt32.dll for use on MS Windows when loading the Windows cert store.

3.1.2   Step by step guide for Upgrading Existing Projects

1. Add the IndySecOpenSSL package to your project's list of required runtime packages.

    For Lazarus, add the package as a "New Requirement" in the Project Manger.

    For Delphi, in the project options, include the package in the list of dependent runtime packages. It is also recommended to select "link to runtime packages".

2. In a GUI project,

    ○ Make a note of which components (e.g. TIdHTTP) have a property linked to a TIdSSLIOHandlerSocketOpenSSL or a TIdServerIOHandlerSSLOpenSSL component.

    ○ Make a note of the property values of any TIdSSLIOHandlerSocketOpenSSL and/or TIdServerIOHandlerSSLOpenSSL components and then delete each such component.

    ○ Replace all of the above with components with, respectively a TIdSecIOHandlerSocketOpenSSL component or a TIdSecServerIOHandlerSSLOpenSSL component from the "IndySec" pallet tab , and set each new component's property values to those recorded for the corresponding (deleted) component.

- Update the components that linked to a TIdSSLIOHandlerSocketOpenSSL or a TIdServerIOHandlerSSLOpenSSL component with a link to the replacement component.

- Ensure that no uses clauses reference the IdSSLOpenSSL unit i.e. remove that reference and replace it with a reference to IdSecOpenSSL if such a reference does not already exist.

3. In a console mode project:

- change (search and replace) all references to the TIdSSLIOHandlerSocketOpenSSL to a reference to the TIdSecIOHandlerSocketOpenSSL

- change (search and replace) all references to the TIdServerIOHandlerSSLOpenSSL class to the TIdSecServerIOHandlerSSLOpenSSL class.

- Replace all uses clauses references to the IdSSLOpenSSL unit to a reference to the IdSecOpenSSL unit.

4. You may also have to add references to the IdSecOpenSSLOptions unit when a TIdSecIOHandlerSocketOpenSSL component or a TIdSecServerIOHandlerSSLOpenSSL component options property is set to a new value.

5. You may also have to add references to the IdSecOpenSSLX509 unit when a TIdSecIOHandlerSocketOpenSSL component OnVerifyPeer Handler references an X.509 certificate.

6. Cleanup and rebuild your project.

> *Note. Advanced users may wish to apply the same approach to GUI projects as described above to console mode projects, while additionally editing each affected .dfm file (Delphi) or .lfm file (Lazarus) to change any class references as described in the first two bullets of list item 3 above.*

3.2   New Projects

3.2.1   Creating The Indy Components

3.2.1.1   GUI Projects

OpenSSL is typically used in support of a protocol such as http. For example, a form that implements an https client will require that a `TIdHTTP` non-visual component is placed on the form alongside a `TIdSecIOHandlerSocketOpenSSL` non-visual component.

Link the `TIdHTTP` IOHandler property to the name of the `TidSecIOHandlerSocketOpenSSL` component.

A similar approach is taken for an https server (TIdHTTPServer) using TIdSecServerIOHandlerSSLOpenSSL.

> *Note: An https server component must have its OnQuerySSLPort handler defined. This should return VuseSSL = true when the APort parameter is set to a port used for SSL/TLS in your application.*

3.2.1.2   Console Mode Projects

In a console mode program, you have to explicitly create an instance of a TIdHTTP component and a `TIdSecIOHandlerSocketOpenSSL`, and set the `TIdHTTP` component's IOHandler property to the name of the TIdSecIOHandlerSocketOpenSSL component. Similarly for an https server/

For example:

TMyApp = class

```
  private
    Fhttp: TidHTTP;
    FSSL: TidSecIOHandlerSocketOpenSSL;
…
  public
   constructor Create;
…
end;

procedure TmyApp.Create;
begin
  inherited Create;
  Fhttp := TidHTTP.Create(nil);
  FSSL :=  TidSecIOHandlerSocketOpenSSL.Create(nil);
  Fhttp.IOHandler := FSSL;
...
end;
```

The examples discussed below (see 3.5) provide guidance on how to use the IndySec components in support of an https client or server.

3.3   Program Link Models

All applications that use the IndySecOpenSSL package must also use the OpenSSL code libraries. These can be provided as static link libraries or dynamic link libraries (e.g. DLLs).

Three link models are supported for linking IndySecOpenSSL applications to these libraries:

- Dynamic Library Load (the default and the approach used in previous versions)
- compile time linkage to a shared (.so or .dll) library (OpenSSL 3.x only)
- compile time linkage to a static library (FPC only with gcc compiled OpenSSL).

3.3.1   Dynamic Library Load

This is the default link model used when no other link model is selected (see below). Under this link model, the OpenSSL dynamic link libraries are loaded at runtime and when needed.

A "Just in Time" approach is used where each API call is initialised to a local proc "loader" function. The intent is that on the first call to a given API function, the dynamic link library is loaded (if not already loaded), and then the actual entry point in the OpenSSL function library is loaded and the API call is set to the loaded entry point. The API function is now called on the user's behalf.

If the call fails to load then it is replaced by a compatibility function (if one exists) - see 3.4. If no such function exists then an exception is raised.

14

3.3.1.1   Library Load and Linux

The Linux library loader searches well known locations for all dynamic libraries (shared objects); Linux package installers ensure that all installed libraries are located in such locations. If you are intending to use the system default OpenSSL libraries (recommended in most cases) then it is sufficient to install the OpenSSL libraries and root certificate store as described in 2.2.1. This applies to both development and target systems.

If your application should use a separately installed version of the OpenSSL libraries i.e. not in a well known location, then the LD_LIBRARY_PATH environment variable may be used to identify the directory where these files may be found.

3.3.1.2   Library Load and Windows

Under Windows and when an application loads a given DLL without specifying an explicit path to the DLL, Windows first searches the same folder as the application executable, then the folders given by the effective PATH environment variable and then searches the \Windows\System32 folder.

For reliable results, the OpenSSL libraries (DLLs) should be located in the same folder as your application's executable (both development and deployment). It is possible to specify an alternative location, but the above is the simplest approach, and the one that should always result in the OpenSSL library that you have selected being used.

> *Note: under Windows, other applications may also have installed their own versions of OpenSSL libraries and these may be a different version to the one you want or, having been optimised for a different use case, omit functions important to your application. It is important to ensure that the correct library is loaded.*

On a development system, you may prefer to locate the OpenSSL libraries at some common location which is then added to the PATH environment variable for applications running under your login.

It is possible for an application to override this default behaviour and to specify an alternative location for the OpenSSL libraries set at runtime (see 4.5).

The example programs provided as part of the IndySecOpenSSL distribution do this so that all examples can use the same copy of the OpenSSL libraries and located in the <indysec>\examples folder (see 3.5).

3.3.2   Compile time linkage to a shared (.so or .dll) library

Under this link model, the OpenSSL dynamic link libraries are loaded by the Operating System and all entry points resolved at program load time. The application has no control over which locations are searched for the link libraries; the Windows default applies.

This link model is used when the OPENSSL_USE_SHARED_LIBRARY symbol is defined at compile time. This defined symbol must be set in the IndySecOpenSSL package options (not the using program).

Lazarus:

In the Package Editor for IndySecOpenSSL, select package options->Custom Options, and click on the "Defines..." button. The dialog illustrated below is shown.
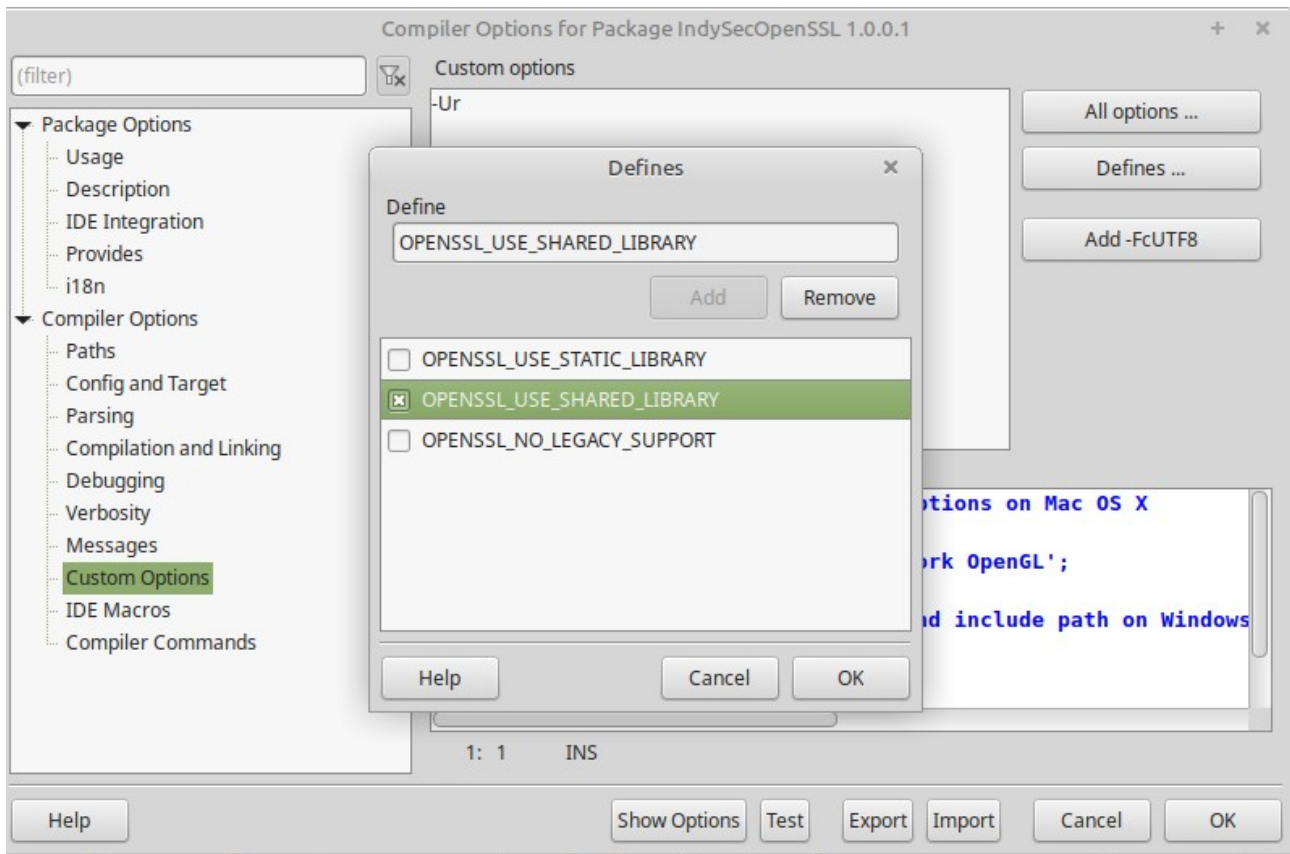
Ensure that the OPENSSL_USE_SHARED_LIBRARY option is selected, and save the package options. You should rebuild the package by selecting "Recompile Clean" from the Package Editor "More" drop down list.

Delphi:

In the Project Options for IndySecOpenSSL.bpl

1. Select "Delphi Compiler" in the left hand window pane.

2. Click on the ellipsis at the end of the "Conditional defines" option in the right hand window pane. The "Conditional defines" dialog should now be shown.

3. Enter OPENSSL_USE_SHARED_LIBRARY into the edit box at the bottom of the dialog and click on the "Add" button.

4. The result should be as illustrated below.
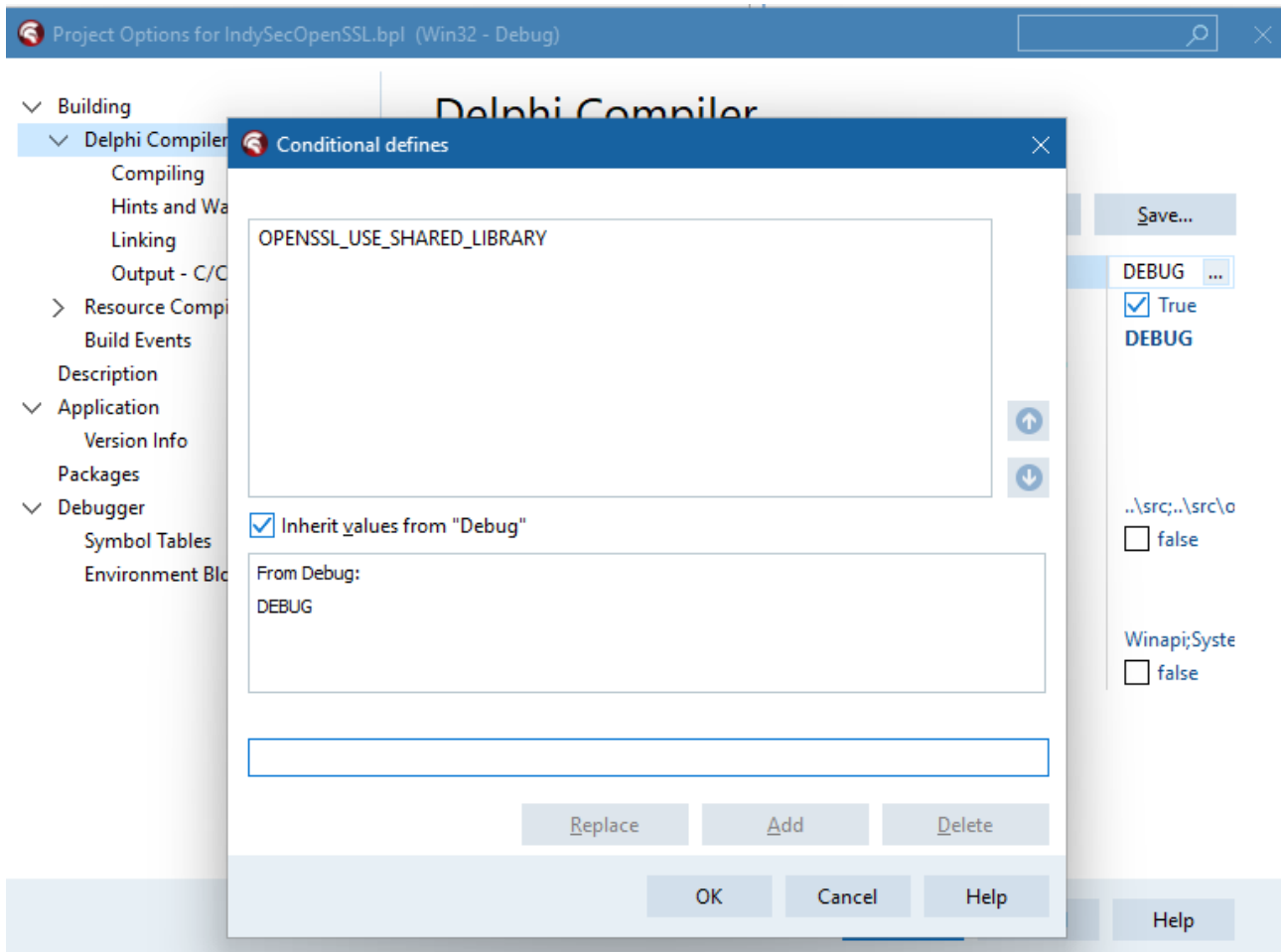
Save the changes, and rebuild the package.

3.3.2.1   Library Load and Linux

See 3.3.1.1.

3.3.2.2   Library Load and Windows

See 3.3.1.2, except that the OpenSSL library location cannot be specified at runtime. That is, the OpenSSL libraries must be located in one of the locations searched by the Windows DLL loader.

3.3.3   Compile time linkage to a static library

> *Note that Static Linking to Static Library at present appears to be reliable for Lazarus/Linux only with gcc generated libssl.a and libcrypto.a. Lazarus/fpc under Windows can have link errors when compiling with gcc generated libraries that include unsupported features, while Delphi has only limited support for static linking..*

Under this link model, the OpenSSL libraries are linked into the program executable and are hence part of the distributed executable. There is no need for the OpenSSL dynamic link libraries to be present on the target system. The downside is a larger executable. However, this model also avoids the risk of loading the wrong or a corrupted version of the OpenSSL dynamic link libraries.

This link model is used when the OPENSSL_USE_STATIC_LIBRARY symbol is defined at compile time. This is set using the same instructions as given above in 3.3.2, but with the

OPENSSL_USE_STATIC_LIBRARY symbol used instead of the

OPENSSL_USE_SHARED_LIBRARY symbol.

Under this model, when compiling and linking an application that uses IndySecOpenSSL, the OpenSSL static link libraries:

- libssl.a

- libcrypto.a

must be available to the linker. For example, on Debian, derivatives (e.g. Ubuntu) you need to install the *libssl-dev* package in order to make these available. Other distros may have a different name for this package.

It is possible to locally compile OpenSSL and use the resulting static link libraries instead. In order to force the linker to search the directory where these libraries are located, add the -Fl option to the package's custom options i.e.

```
-Fl<path to directory>
```

It may also be necessary to uninstall the libssl-dev package to ensure that there is no confusion as to which version of the libraries is used.

3.4   Compatibility Functions

The purpose of "compatibility functions" is to enable continuing support of older versions of OpenSSL (e.g. 1.0.2) for deployments that cannot readily move forward to the latest version. They apply only to the Dynamic Library Load link model and replace and emulate a limited number of OpenSSL 3.x API calls using legacy OpenSSL API calls.

The defined symbol OPENSSL_NO_LEGACY_SUPPORT may also be set at compile time and applies to Dynamic loading only. If set, no compatibility functions are compile in to the executable. Only 3.0 or later API calls may be used. This can be used to reduce code size and avoid a risk of using an unsupported OpenSSL library.

3.5   Examples

The software distribution contains examples implementations of both https clients and servers. These are used to both demonstrate usage and to provide test programs. By default, all examples are configured to the use the Dynamic Load and Link Model.

These examples provide useful templates covering most OpenSSL use cases.

The console mode examples can be run without first installing IndySecOpenSSL into the IDE. The GUI examples are best run after installation. If IndySec is not installed then the IDE will complain that the project contains unknown components when the form is opened.

- On Delphi, the <indysec>\examples\AllExamples.projgroup provides a simple way to access the package and the two console mode examples and to then build and run each example.

- On Lazarus, you can run the console mode examples prior to installation by first opening and optionally compiling the <indysec>/lazarus-fpc/IndySecOpenSSL.lpk package. You can then open and run each console mode example program.

Both console mode examples take command line arguments, including an alternative means of locating the OpenSSL Dlls.

On Windows, it is recommended to install the OpenSSL libraries in the

- <indysec>/\<examples

folder. All example programs are configured to first attempt to load the OpenSSL DLLs from this folder, and then to use the Windows default load strategy (see 2.2.2). This is achieved by setting the OpenSSLPath to

..\..;

That is a semi-colon separated list where the second list item is empty.

It is recommended to install both 32-bit and 64-bit DLLs into the examples folder. By default Delphi expects 32-bit DLLs and Lazarus expects 64-bit DLLs, but both can work with either. The DLL names are typically:

- libssl-3.dll
- libcrypto-3.dll
- libssl-3-x64.dll
- libcrypto-3-x64.dll

Note. Different sources for these DLLs may choose different names.

3.5.1 An https Client

The <indysec>/examples/openssl-client contains both GUI and Console mode (CLI) variants of an openssl-client, with both delphi and lazarus/fpc project files. These are respectively contained in the "guI" and "cli" subdirectories. i.e.

- <indysec>/examples/openssl-client/cli
- <indysec>/examples/openssl-client/gui

In each case, the objective is that same. That is to illustrate:

1. An https Get on a test server that returns a simple message containing "Success!"

2. The "Get" is performed twice. The first time there is no verification of the remote server's certificate. The second time, the remote server's certificate is verified using the local trusted store of root certificates.

3. Progress information is reported using the OnStatusInfo event handler.

4. For the second "Get" the result of the verification is reported including the validated certificate(s). Typically, more than one certificate is verified. That is the entire certificate chain is shown.

5. The console mode program reports to the console. The GUI program uses a TMemo on the main form to report progress.

The console mode example also takes program arguments:

Usage: [-h] [-n] [-l <cacerts dir>] [-L] [OpenSSL lib dir]

-h	Shows the program help information
-n	When present forces a prompt to "continue" on program completion. This is useful under Windows when running from the IDE, as it avoids the command line window being closed on completion, and requires the user to press the enter button in order to close the window.
-L	Linux only. Scans well known locations for the ca certs directory. This is useful when using OpenSSL libraries that have been locally compiled and compiled with a default ssl ca certs dir that is not present. The well known locations are locations that are distro dependent.
-l <ca certs dir>	Explicitly sets the directory containing a trusted list of ca certificates.
<openssl lib dir>	When given as the last argument, this is used as the directory from which the OpenSSL libraries are loaded. A useful alternative to setting the Working Directory when a common location is used for the OpenSSL DLLs.

3.5.1.1 Example Results from the Console Mode program

This example output was copied obtained when running the program with no arguments under Linux Mint 22.2.

```
Using OpenSSL 3.0.13 30 Jan 2024, OpenSSLDir: /usr/lib/ssl
Link Model: Dynamic linking at run time
LibCrypto:
LibSSL:
Working Directory =
/home/tony/SoftwareDev/MWA/IndySec/examples/openssl-client/cli

Getting https://test.mwasoftware.co.uk/openssltest.txt with no verification

Status Info: SSL status: "before SSL initialization"
Status Info: SSL status: "before SSL initialization"
Status Info: SSL status: "SSLv3/TLS write client hello"
Status Info: SSL status: "SSLv3/TLS write client hello"
Status Info: SSL status: "SSLv3/TLS read server hello"
Status Info: SSL status: "TLSv1.3 read encrypted extensions"
Status Info: SSL status: "SSLv3/TLS read server certificate"
Status Info: SSL status: "TLSv1.3 read server certificate verify"
Status Info: SSL status: "SSLv3/TLS read finished"
Status Info: SSL status: "SSLv3/TLS write change cipher spec"
```

```
Status Info: SSL status: "SSLv3/TLS write finished"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: Cipher: name = TLS_AES_256_GCM_SHA384; description =
TLS_AES_256_GCM_SHA384        TLSv1.3 Kx=any      Au=any    Enc=AESGCM(256)
Mac=AEAD
; bits = 256; version = TLSv1.3;
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSLv3/TLS read server session ticket"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSLv3/TLS read server session ticket"
Status Info: SSL status: "SSL negotiation finished successfully"
Using SSL/TLS Version TLSv1.3 with cipher TLS_AES_256_GCM_SHA384
Remote Source returned:
Success!

Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Getting https://test.mwasoftware.co.uk/openssltest.txt with verification

Using certs from /usr/lib/ssl/certs
Status Info: SSL status: "before SSL initialization"
Status Info: SSL status: "before SSL initialization"
Status Info: SSL status: "SSLv3/TLS write client hello"
Status Info: SSL status: "SSLv3/TLS write client hello"
Status Info: SSL status: "SSLv3/TLS read server hello"
Status Info: SSL status: "TLSv1.3 read encrypted extensions"

Client Side Verification
Root Certificate verification succeeded

X.509 Certificate Details
Subject: /C=US/O=Internet Security Research Group/CN=ISRG Root X1
Issuer: /C=US/O=Internet Security Research Group/CN=ISRG Root X1
Not Before: 4-6-15 12:04:38
Not After: 4-6-35 12:04:38


Client Side Verification
Remote Certificate verification succeeded

X.509 Certificate Details
Subject: /C=US/O=Let's Encrypt/CN=E8
Issuer: /C=US/O=Internet Security Research Group/CN=ISRG Root X1
Not Before: 13-3-24
Not After: 12-3-27 23:59:59


Client Side Verification
Remote Certificate verification succeeded

X.509 Certificate Details
Subject: /CN=mwasoftware.co.uk
Issuer: /C=US/O=Let's Encrypt/CN=E8
Not Before: 3-10-25 15:19:39
Not After: 1-1-26 14:19:38
```

```
Status Info: SSL status: "SSLv3/TLS read server certificate"
Status Info: SSL status: "TLSv1.3 read server certificate verify"
Status Info: SSL status: "SSLv3/TLS read finished"
Status Info: SSL status: "SSLv3/TLS write change cipher spec"
Status Info: SSL status: "SSLv3/TLS write finished"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: Cipher: name = TLS_AES_256_GCM_SHA384; description =
TLS_AES_256_GCM_SHA384        TLSv1.3 Kx=any     Au=any   Enc=AESGCM(256)
Mac=AEAD
; bits = 256; version = TLSv1.3;
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSLv3/TLS read server session ticket"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSLv3/TLS read server session ticket"
Status Info: SSL status: "SSL negotiation finished successfully"
Using SSL/TLS Version TLSv1.3 with cipher TLS_AES_256_GCM_SHA384
Remote Source returned:
Success!

Status Info: SSL status: "SSL negotiation finished successfully"
Status Info: SSL status: "SSL negotiation finished successfully"
```

> *Note. The empty LibSSL and LibCrypto lines indicate that the system defaults are used. An explicit path is given otherwise.*

3.5.2   An https Server

The <indysec>/examples/openssl-server contains a Console mode (CLI) variants of an OpenSSL server, with both delphi and lazarus/fpc project files. These are respectively contained in the "gui" and "cli" subdirectories. i.e.

- <indysec>/examples/openssl-server/cli
- <indysec>/examples/openssl-server/gui

In each case, the objective is that same. That is to illustrate:

1. Getting a URL from a local server. The program contains both an https client and an https server. The client runs in the main thread, the server in a second thread.

2. Get without client verification, and then

3. Get with client verification

4. Progress information is reported using the OnStatusInfo event handler.

5. A private PKI is used with the certificates provided in the

   ○ <indysec>/examples/openssl-server/cacerts, and

   ○ <indysec>/examples/openssl-server/certs directories.

6. The program illustrates use of a private PKI. The necessary certificates are provided with the example. However, the script <indysec>/examples/createpki.sh can be used to

regenerate the certificates and provides an example of how they were generated using the *openssl* command line utility - usually provided with the OpenSSL libraries.

Note. The GUI example sets the certificate paths at runtime rather than using the object inspector. This is help portability between Linux and Windows environments. Most use cases will prefer to use the object inspector.

4

Reference

4.1 TIdSecIOHandlerSocketOpenSSL

This class is declared in the IdSecOpenSSL unit, and is used to support protocol clients.

4.1.1 Properties

SSLOptions	Used to set various options. See TIdSecOptions (section 4.3)
OnBeforeConnect	Called immediately before the TLS Handshake is performed. Note. Cannot be used to update SSLOptions.
OnStatusInfo	Called to report significant TLS protocol events. Provides a text description that may be used to indicate progress to the user.
OnStatusInfoEx	As above, but provides more detailed debugging information.
OnGetPassword	Called when decoding a passphrase protected X.509 certificate. Returns the passphrase either from some local information source or obtained by prompting the user.
OnGetPasswordEx	As above but includes the IsWrite: boolean parameter. This indicates whether the passphrase is to be used to decode (false) or encode (true) the certificate.
OnVerifyPeer	Called to report the result of peer certificate verification. The event includes the X.509 certificate provided (object class TIdX509), its

	verification status and on failure additional error codes.
	The TIdX509 class declaration may be found in the IdSecOpenSSLX509 unit and includes string type public properties that may be used to provide a textual representation of the certificate.

4.2 TIdSecServerIOHandlerSSLOpenSSL

This class is declared in the IdSecOpenSSL unit, and is used to support protocol servers.

4.2.1 Properties

Identical to TIdSecIOHandlerSocketOpenSSL.

4.3 TIdSecOptions

The same TidSecOptions class is used for both clients and servers.

This class is declared in the IdSecOpenSSLOptions unit.

4.3.1 Properties

RootCertFile	The path to a file providing the X.509 root certificate for certificate verification (optional for a client). May be either PEM or p12[1] encoded. This is the root certificate used to sign the client or server's certificate.
CertFile	The path to a file providing the X.509 certificate for the local http entity (optional for client). May be either PEM or p12 encoded.
KeyFile	The path to the file providing the private key corresponding to the above X.509 certificate. Required when CertFile present. May be either PEM or p12 encoded.
DHParamsFile	An "X.509 Diffie Hellman parameters file" (or DH parameters file). This is a file containing the public prime (P) and base (G) values for the Diffie-Hellman key exchange algorithm., These parameters are typically generated using tools like `openssl dhparam`, stored in the PEM format (e.g., as `dh4096.pem`), and used by servers to create temporary DH key pairs during the TLS handshake. Optional.

---

[1]A PKCS12 encoded certificate file (extension .p12 or .pfx) is a passphrase protected bundle comprising the entities X.509 certificate, root certificate and encryption key. The RootCertFile, CertFile and KeyFile are typically set to the same p12 file.

Method	Deprecated and ignored. In earlier versions this was used to select the SSL/TLS version used. Modern OpenSSL libraries always negotiate the most secure TLS version regardless of this setting. Default: sslvTLSv1_3
SSLVersions	Deprecated. Lists the permissible SSL/TLS protocol versions. With modern OpenSSL libraries this can be used to select the minimum acceptable version only. Default. sslvTLSv1_2,sslvTLSv1_3.
Mode	This is used to identify whether a client or server is intended. Default: - TIdSecIOHandlerSocketOpenSSL: sslmClient - TIdSecServerIOHandlerSSLOpenSSL: sslmServer Available values: sslmUnassigned, sslmClient, sslmServer, sslmBoth
VerifyMode	Determines the action taken when verifying an X.509 certificate. Value is a set of: sslvrfPeer: **Server mode:** the server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. **Client mode:** the server certificate is verified. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. If no server certificate is sent, because an anonymous cipher is used, sslvrfPeer is ignored. sslvrfFailIfNoPeerCert: **Server mode:** if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a "handshake failure" alert. Only valid when sslcrfPeer is also present in the set. **Client mode:** ignored sslvrfClientOnce:

	Server mode: only request a client certificate on the initial TLS/SSL handshake. Do not ask for a client certificate again in case of a renegotiation. Only valid when sslcrfPeer is also present in the set. **Client mode:** ignored Empty set: **Server mode:** the server will not send a client certificate request to the client, and hence the client will not send a certificate. **Client mode:** if not using an anonymous cipher (by default disabled), a server certificate is required and which will be verified on receipt.
VerifyDepth	Determines the maximum **depth** for the certificate chain verification that shall be allowed for X.509 certificate verification.
VerifyDirs	This property should be empty when the system default root certificate store is used. It may be set to the path of a directory containing a set of trusted CA certificates, one per file. This should be used only when the root certificate store is in a location other than that compiled into OpenSSL or an alternative is to be used. Alternative, when a Private PKI is used, the VerifyDirs should contain the Private PKI's root certificate(s).

UseSystemRootCertificateStore	Windows:
	
	Depends on whether or not the symbol OPENSSL_DONT_USE_WINDOWS_CERT_STORE is defined at compile time (in the package options)
	
	By default:
	
	    When true specifies that the Windows root certificate store is used as the set of trusted CA Certificates in addition to any root certificates in the VerifyDirs folder.
	
	If the symbol OPENSSL_DONT_USE_WINDOWS_CERT_STORE is defined at compile time then
	
	    When true specifies that the compiled in (to the OpenSSL libraries) default location is used to load root certificates.

	Linux: When true specifies that the compiled in (to the OpenSSL libraries) default location is used to load root certificates. In all cases, when false only root certificates in the VerifyDirs folder are used.
CipherList	Sets the list of available ciphers (TLSv1.2 and below only). If left empty then the system default is used.
TLS1_3_CipherList	Sets the list of available ciphers (TLS1.3 and later). If left empty then the system default is used.

4.4   The IOpenSSL Interface

The IdSecOpenSSLAPI unit provides two Pascal COM interfaces which provide information and option selection for the OpenSSL Library. This interface is available for all link strategies and the interface is accessed using the function:

```
function GetIOpenSSL: IOpenSSL;
```

The interface is declared as:

```
  TOpenSSL_LinkModel = (lmDynamic, lmShared, lmStatic);

  IOpenSSL = interface
  ['{aed66223-1700-4199-b1c5-8222648e8cd5}']
    function GetOpenSSLPath: string;
    function GetOpenSSLVersionStr: string;
    function GetOpenSSLVersion: TopenSSL_C_ULONG;
    function GetLinkModel: TopenSSL_LinkModel;
    function Init: boolean;
  end;
```

`function GetOpenSSLPath: string;`	Static Linking: Returns the OpenSSL installation path as compiled into the OpenSSL library. Dynamic Loading: Returns - The same as above when the library is loaded - the specified OpenSSL installation path, otherwise.
function GetOpenSSLVersionStr: string;	Returns the OpenSSL library version string. e.g

	"OpenSSL 3.2.0 23 Nov 2023"
function GetOpenSSLVersion: TOpenSSL_C_ULONG;	Returns the OpenSSL library version as an integer as defined by the OpenSSL documentation.
`function GetLinkModel: TOpenSSL_LinkModel;`	Returns the Link Model used. Values: lmDynamic, lmShared, lmStatic
function Init: boolean;	Called to initialise the OpenSSL library prior to use. It is optional for dynamic linking (implicitly called on library load). It is optional for static linking (implicitly called on unit initialisation).

4.5   The IOpenSSLDLL Interface

This interface is only available when the API is configured at compile time for dynamic library loading. The interface is accessed using the function:

```
function GetIOpenSSLDDL: IOpenSSLDLL;
```

This function returns "nil" if the interface is not available. This may be used as a runtime test to determine if dynamic library loading if been configured.

Note The OpenSSLAPI unit only declares the constant

OpenSSL_Using_Dynamic_Library_Load = true;

when configured at compile time for dynamic library loading. The constant is not declared otherwise. This feature may be used as a compile time test for the dynamic library loading strategy e.g.

{$if declared(OpenSSL_Using_Dynamic_Library_Load)}

…{$ifend}

Similarly, the constant

OpenSSL_Using_Shared_Library = true;

is only declared when using the share library link model

The interface includes the IOpenSSL interface and is declared as:

```
  IOpenSSLDLL = interface(IOpenSSL)
    procedure SetOpenSSLPath(const Value: string);
    function GetSSLLibVersions: string;
    procedure SetSSLLibVersions(AValue: string);
    function GetSSLBaseLibName: string;
    procedure SetSSLBaseLibName(AValue: string);
    function GetCryptoBaseLibName: string;
    procedure SetCryptoBaseLibName(AValue: string);
    function GetAllowLegacyLibsFallback: boolean;
    procedure SetAllowLegacyLibsFallback(AValue: boolean);
    function GetLibCryptoHandle: TLibHandle;
    function GetLibSSLHandle: TLibHandle;
    function GetLibCryptoFilePath: string;
    function GetLibSSLFilePath: string;
    function GetFailedToLoadList: TStrings;
    function Load: Boolean;
    procedure Unload;
    function IsLoaded: boolean;
    property SSLLibVersions: string read GetSSLLibVersions
                                    write SetSSLLibVersions;
    property SSLBaseLibame: string read GetSSLBaseLibName
                                   write SetSSLBaseLibName;
    property CryptoBaseLibName: string read GetCryptoBaseLibName
                                       write SetCryptoBaseLibName;
    property AllowLegacyLibsFallback: boolean read GetAllowLegacyLibsFallback
                                      write SetAllowLegacyLibsFallback;
end;
```

procedure SetOpenSSLPath(const Value: string);	This sets the OpenSSLPath used to locate the OpenSSL library modules (e.g. libcrypto.so). It needs to be set when OpenSSL has not been installed in a default location and/or multiple versions of OpenSSL have been installed on the same system. The OpenSSLPath may be empty (default), contain a single path, or a colon (Unix) or semi-colon (Windows) separated list of paths to try in order.
function GetSSLLibVersions: string; procedure SetSSLLibVersions(AValue: string);	This is a colon separated (Unixes) or semi-colon separated (Windows) ordered list of OpenSSL version numbers that can be used as suffices for the OpenSSL library modules (e.g. for libcrypto-3-x64.dll, the suffix is '-3-x64'). When searching for the OpenSSL library, the loader searches the default (or specified) OpenSSLPath for OpenSSL libraries using these suffices in turn. Unix: defaults to '.3:.1.1:.1.0.2:.1.0.0:.0.9.9:.0.9.8:.0.9.7:.0.9.6'

	Note includes legacy versions.
	Windows defaults to
	'-3-x64;-1-x64;' (64 bit)
	'-3;-1;' (32 bit)
	Changing the libversions will automatically unload the ssl and crypto libraries if currently loaded.
function GetSSLBaseLibName: string; procedure SetSSLBaseLibName(AValue: string);	These are used to respectively get and set the basename for the SSL dynamic library. Defaults to 'libssl'. Changing the base name will automatically unload the ssl and crypto libraries if currently loaded.
function GetCryptoBaseLibName: string; procedure SetCryptoBaseLibName(AValue: string);	These are used to respectively get and set the basename for the crypto dynamic library. Defaults to 'libcrypto'. Changing the base name will automatically unload the ssl and crypto libraries if currently loaded.
function GetAllowLegacyLibsFallback: boolean; procedure SetAllowLegacyLibsFallback(AValue: boolean);	These are used to respectively get and set the AllowLegacyLibsFallback flag. This is only used when running under Windows. If set and no OpenSSL libraries have been found when searching for them using the Base Library names and libversions, then the loader will attempt to load the OpenSSL libraries using the legacy library names 'libeay32' and 'ssleay32'. Defaults to 'false'. Changing this flag will automatically unload the ssl and crypto libraries if currently loaded.
function GetLibCryptoHandle: TLibHandle;	After a successful library load, this returns the value of the internal handle to libcrypto (internal use only recommended).
function GetLibSSLHandle: TLibHandle;	After a successful library load, this returns the value of the internal handle to libssl (internal use only recommended).
function GetLibCryptoFilePath: string;	After a successful library load, this returns the

	path to the loaded libcrypto library. (note: may be empty if default library loaded).
function GetLibSSLFilePath: string;	After a successful library load, this returns the path to the loaded libssl library. (Note: may be empty if default library loaded).
function GetFailedToLoadList: TStrings;	After a successful library load, this returns a list of API call names that failed to load at library load time. Note: only applies to a small number of functions that are not suitable for "just in time" loading.
function Load: Boolean;	Explicitly loads the library if not already loaded and returns "true" on successful load.
procedure Unload;	Explicitly unloads the library if current loaded.
function IsLoaded: boolean;	Returns true if the OpenSSL library has been successfully loaded.

Appendix A. The Transport Layer Security Protocol

The Transport Layer Security (TLS) protocol is defined in RFC 8446, and is intended for use with stream based transport services (e.g. TCP). TLS itself is an onward development of the Secure Socket Layer (SSL) protocol originating in Netscape Navigator and for use in support of secure website communication.

It is complemented by a datagram version, the Datagram Transport Security (DTLS) . DTLS is defined in RFC 9147.

A.1. Overview

The original implementation scenario for TLS/SSL is that of a Web Browser Client accessing a Web Server using the Hypertext Transfer Protocol (https). When used with no security, http is a simple request/response protocol using a stream based transport service (e.g. TCP) to request a resource from a website and to receive the resource in response. In more advanced scenarios, an http request may include a resource and receive a simple acknowledgement in return, or another resource.

When supported by the Web Server, a Web Client may decide to secure this communication by using TLS. The TLS protocol slots in between http and the TCP protocol without either needing to be modified. The use of http with security is typically identified as https, and https is often used as the *scheme* part of a URL in order to force the use of TLS.

This model of communication is fundamental to the design of the TLS. TLS is typically used to:

- Authenticate a Website to a Client
- Protect the integrity of the data exchange
- Protect the confidentiality of the data exchange by using encryption
- Prevent man-in-the-middle attacks

Optionally, a Website may additionally require a Client to identify and authenticate themselves before accessing a protected resource. This identification and authentication can be supported by TLS – although often other mechanisms (e.g. passwords, etc.) are used that are not part of TLS but nevertheless make use of the confidentiality and integrity provide by TLS.

TLS based Client Authentication typically takes place at the start of a session, but can be delayed until later.

A.1.1. The Structure of a TLS Session

TLS is a framework for secure communications. It describes a protocol for managing Key Agreement and Authentication, the protected exchange of application data and re-keying. However, the algorithms used for protecting application data, authentication and Key Agreement are outside of the scope of the specification. It is agnostic as to the algorithms used, but does provide the means by which Client and Server can negotiate dynamically which are used for a given TLS Session.

There is a strong tilt towards public key based authentication supported by X.509 Certificates. However, even this is not mandatory.

A.1.2. The TLS Session

All application data exchange takes place within a TLS Session. A TLS Session starts when a TCP connection starts and continues until the TCP connection is terminated. TLS is the user of the TCP connection and all application communication is performed through TLS. TLS Sessions can be resumed at the start of a new TCP connection either after an initial session has closed or in parallel to existing sessions.

Data is exchanged by TLS as a series of typed blocks with a length specified in the block header. Blocks may be either Plain Text (unencrypted) or Cipher Text (encrypted). Blocks are first formatted as Plain Text. Once record protection has been engaged (i.e. Key Agreement has completed) then Plain Text Blocks are transformed into Cipher Text Blocks when they are transmitted. Before Key Agreement has been completed, Blocks are sent in their original Plain Text format. Record Protection must start before Application Data is exchanged.

TLS Plain Text Blocks are classified as "Handshake", "Change Cipher Spec", "Alert" and "Application Data". Handshake blocks comprise messages used to perform Key Agreement and Authentication, and any other negotiation that needs to take place at the start of a session.

Alert blocks signal errors and session closure. Application Data Blocks are used to transfer Application Data. Change Cipher Spec Blocks are for legacy use only.

A.1.3. The Handshake Protocol

This protocol is used to support Key Exchange and Authentication. The following illustration of the messages exchanged is copied from the RFC.

```
Client                                          Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*        -------->
                                     ServerHello        ^ Key
                                     + key_share*        | Exch
                                     + pre_shared_key* v
                                     {EncryptedExtensions}  ^  Server
                                     {CertificateRequest*}  v  Params
                                             {Certificate*}  ^
                                      {CertificateVerify*}  | Auth
                                                 {Finished}  v
                                <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

  * => optional

The protocol is structured as a set of messages (e.g. ClientHello, ServerHello, Certificate, etc.) which may be sent individually or, when required, concatenated as a single block – known as a "Flight" in TLS. Each message comprises a basic set of parameters plus extensions. As the protocol has developed through successive versions, most functionality is now provided through extensions while many of the basic parameters are labelled "legacy" and ignored. Extensions are registered through IANA with a number of extension identifiers reserved for private use. This makes TLS a readily extensible protocol within its basic framework.

The basic principle of the handshake protocol is to first agree a shared secret and from which cryptographic keys can be generated before authenticating the server to the client and, optionally the client to the server:

1. A Client initiates a TCP connection with the Server and sends a ClientHello message immediately afterward the connection is initiated.

   In the message header, the ClientHello identifies the set of cipher suites supported. They are thus negotiable and this allows the encryption and integrity check algorithms to be specified and evolve separately from the TLS protocol itself. In versions of TLS before 1.3, a ClientHello was also able to offer data compression. However, this was withdrawn in TLS 1.3.

   The *key_share* extension provides Key Exchange information for use in a Diffie-Hellman key agreement scheme. There is no fixed key agreement scheme with both standard and Elliptic Curve Diffie-Hellman referenced by the TLS RFC. But other schemes may be supported.

   The *signature_algorithms* extension provides a list of digital signature algorithms supported by the Client for use in authentication.

   The *psk_key_exchange_modes* and the *pre_shared_key* and used for session resumption and are discussed below.

Note that there is no client identification given in the ClientHello. In older version of the TLS there was also no means for a ClientHello to identify the server it wanted to use. This was fixed in RFC6066 by a ServerName extension. This was an important addition in support of virtual web servers. Early use of TLS/SSL required that each secure web server had a unique IP Address. This extension allowed multiple secure (virtual) web servers to share a single IP Address.

2. The server responds with either a ServerHello (illustrated above) or a HelloRetryRequest message. This latter message is used when it is necessary for the Client to be requested to resend the ClientHello with additional information e.g. in a cookie exchange – see below).

   The ServerHello header includes the list of cipher suites supported and selected by the server and for use on this session, and completes the negotiation of the cipher suites. It also provides its own *key_share* extension to provide its key exchange information. Both Client (when it receives the ServerHello) and the Server are now able to determine the shared secret (using Diffie Hellman) and hence derive the keying material for the rest of the session.

3. Several message blocks can be concatenated with a ServerHello and the first of these is the *EncryptedExtensions* block. This is used to convey information returned by the server that must be protected. Given that keying material has been agreed with the exchange and receipt of the ServerHello, it is possible for the Server to encrypt this data and for the Client to decrypt it.

   Examples of protected extensions include the *server_name* extension confirming the server's name and *early_data* allowing the exchange of limited application data at this point of the session. Generally, any extension vulnerable to a man-in-the-middle attack is encoded in this message.

4. The handshake now moves on to authentication and any or all of the *CertificateRequest, Certificate* and *CertificateVerify* messages may follow the *EncryptedExtensions*. Note that X.509 certificate based authentication is usually used, but this is not required. Raw Public Keys are also supported by the industry standard.

   a. A *CertificateRequest* message is sent by a Server to a Client to request the Client's own certificate. This is in the rare (in web servers) case where a Client needs to both identify itself to the Server and to authenticate itself to the Server. This message will often include a *certificate_authorities* extension allowing the Server to tell the Client which CAs it accepts. For example, a web browser may have several client certificates installed signed by different CAs. This extension allows the Client to select the correct user certificate to provide to the Server.

   b. The *certificate* message must be provided "whenever the agreed-upon key exchange method uses certificates for authentication." It will by default contain the Server's X.509 DER encoded certificate and, if necessary, the certificate path (i.e. a chain of certificates leading back to a common point of trust).

      An alternative encoding (e.g. Raw Public Keys) may be used if the Server has specified this using a *server_certificate_type* extension (RFC 7250) contained in the EncryptedExtensions message, and when the Client has

indicated support for the encoding by also including a *server_certificate_type* extension in its ClientHello.

    c. The *CertificateVerify* message provides a Digital Signature over the handshake exchange, using the agreed Digital Signature Algorithm and the public key provided by the certificate (if using X.509 certificates). This may be used by the Client to authenticate the Server.

5. The Server completes its Authentication messages with a *Finished* message. This is a simple message indicating the end of the message block: it contains an HMAC over the message exchange thus binding the certificate identity verification to the shared secret agreed during Key Establishment.

6. The Client receives the ServerHello through to Finished message as described above and then proceeds to:

- Use the ServerHello to complete the Key Agreement
- Decode the EncryptedExtensions and apply them
- Verify the Certificate provided by the Server
- Authenticate the Server and the binding of the authentication to the Key Agreement.

On success, the Client returns its own Finished message to the Server and secure application data exchange may take place.

7. If the Server had requested a Client Certificate then, prior to the Finished Message, the Client will send its DER encoded X.509 certificate and a CertificateVerify message that may be used by the Server to authenticate the Client.

A.1.4. Certificate Caching

The Certificate Message is likely to be 1 to 2Kbytes in size and may include further certificates when a key path needs to be provided. RFC 8879 does provide a means to compress certificates (e.g. using zlib). However, even a compressed certificate will be quite large given that DER encoded certificates are not necessary very compressible.

The mandatory nature of the certificate message and the use of an X.509 certificate has performance implications for slow speed Datalinks. However, RFC 7924 describes a TLS extension to support certificate caching and which can be used to reduce the overhead of using X.509 certificates.

A.1.4.1. Client Caching of a Server Certificate

When a Client supports certificate caching and has cached a copy of the Server's certificate, it signals this by adding a *cached_info* extension (of type "cert") to its ClientHello. This extension may be repeated if more than one certificate is cached for the same Server.

The extension includes the "fingerprint" of the cached certificate. This is 32 byte SHA-256 hash of the DER encoded certificate and should be a unique identification for the cached certificate.

The Server inspects this list and if one of the cached certificates listed matches its own certificate then it will use this certificate and signals this choice by replacing the X.509 certificate in its Certificate Message with the fingerprint of the selected X.509 certificate, rather than the full certificate. If the ClientHello does not provide the fingerprint of a valid or otherwise unknown certificate then the Server returns its X.509 certificate as normal.

In either case, this is followed by a CertificateVerify message containing a Digital Signature signed using the private key associated with the selected Server Certificate.

Note: so that the Client can determine the format of the Certificate Message it receives from the Server, when the message contains a fingerprint, it must be preceded by a cached_info extension of type "cert" in its ServerHello.

A.1.5. Certificate Validation

A Client must validate a Server Certificate prior to use and likewise, a Server must validate and Client Certificate.

In order to validate a certificate, the certificate signature must be verified using a trusted CA and, the certificate must be checked against the CA's current Certificate Revocation List (CRL), to ensure that it has not been revoked (e.g. because it is known to be compromised).

This forces (e.g.) Web Clients to:

- Keep Root Certificates for all or most of the CAs that a Server might use.
- Be provided with Server Certificate's Certificate Path when it uses a less common CA.
- Keep up-to-date copies of each CA's revocation list (CRL) so that it can identify any certificates revoked before their expiry date.

A Client can signal in its ClientHello, the list of CAs that it trusts, by including a *certification_authorities* extension. This means that a Server need only provide a Certificate Path (when it has one) when its own CA is not trusted directly by the Client. This limits both the number of CAs that a Client need trust and avoids unnecessary transmission of a (long) certificate path. However, it does nothing to help with identifying revoked certificates.

A.1.5.1. OCSP

The Online Certificate Status Protocol (OCSP) is a simple request/response protocol defined in RFC 6960 and which can be used to support certificate validation. OCSP is used by either a TLS Client or Server to query a trusted OCSP Server to determine if it can trust a given certificate.

The request sent to an OCSP Server identifies the certificate to be validated using hashes of the certificate issuer name and the issuer public key, and the certificate serial number. The first two are used to identify the CA, while the serial number identifies the certificate to the CA. The OCSP Server can then verify that the certificate is current and has not been revoked. It then returns the status of the certificate (Good, Revoked or Unknown) and signs the response.

The requesting TLS Client or Server has to verify the Digital Signature on the response and for this needs a trusted certificate for the OCSP Server. If the response is "good", and the Digital Signature is also verified, then it can proceed to authenticate its peer.

Note the TLS Client or Server only needs to have the certificate for the OCSP Server. It does not need to have the CA Root Certificate or any certificate path, or any CRLs for the certificate to be validated. This is all offloaded to the (trusted) OCSP Server.

A.1.5.2. OCSP and TLS

A direct interaction with an OCSP Server is an extra overhead that may cause problems for Clients on low bandwidth links and may also cause a high load on the OCSP Server. Indeed, OCSP Server response time could become a drag on the performance of (e.g.) high volume web servers. To avoid these problems, RFC 6066 has incorporated the OCSP exchange within TLS resulting in a technique known as "OCSP Stapling". This has later been updated to be an integral part of TLS 1.3.

The basic idea is that a TLS Client additionally requests that the TLS Server obtains and provides an OCSP Response for its certificate and provides this with its certificate. That is, the Server interacts with an OCSP Server trusted by the TLS Client and returns the response received from the OCSP Server to the TLS Client with the Server Certificate.

This avoids the need for the TLS Client to do this and keeps the protocol overhead (TLS Client side) to a minimum. This also allows TLS Servers to cache OCSP Responses for their certificates and return the same response each time it is requested, thus minimising the impact on OCSP Servers. The response only needs to be refreshed when it has outlasted its validity time.

Note that the TLS Server does not need to validate the Digital Signature on the OCSP Response. It simply passes it on to its Client.

A.1.5.3. Client requested OCSP Stapling

When a TLS Client wants to use OCSP Stapling, it adds a "status_request" extension to its ClientHello and includes with it a list of OCSP Responder IDs indicating which OCSP Servers it trusts.

The TLS Server then includes the requested OCSP Response as an extension to its Certificate Message. This includes the OCSP Server's signature, allowing the Client to validate the OCSP Response and may then assert that the Server Certificate is valid and acceptable for use.

A.1.5.4. Server Requested OCSP Stapling

It is also possible for a Server to request that a Client presents an OCSP Response with its Certificate. This is by sending an "empty" *status_request* extension in its CertificateRequst message (RFC8446 para 4.4.2.1). However, this means that the Server has no means to tell the Client which OCSP Responders that it trusts. There is no clear reason why this is so, but this is how the protocol is defined. The Client must know, by other means, which OCSP Responder to use.

Note that in RFC 6066 an empty *status_request* extension is specified as implying that the list of trusted responders is known *a priori*.

When OCSP Stapling requested, a TLS Client returns the OCSP Response as an extension to its Certificate Message.

A.1.5.5. OCSP Stapling and Certificate Caching

The TLS protocol does not allow an OCSP Response to be included in a Server's Certificate Message when the message contains a fingerprint instead of an X.509 certificate. There is also no Certificate Message returned by a Client when certificates are cached. Hence, OCSP stapling cannot be used when certificate caching is used.

When OCSP stapling is used, it is implicit that a certificate should remain in a certificate cache only for as long as it is considered valid and has not exceeded its OCSP update time limit.

A.1.6. The Cookie Exchange

In TLS, the cookie exchange is used by a Server to provide a return routablity check prior to processing the Client's Key Exchange Information. This is to avoid a potential DoS attack where a malicious client sends a ClientHello from either a spoofed IP Address or simply does not bother to respond to any Server Responses. The purpose of such a (rogue) ClientHello is to make the Server incur the processing overhead of a Diffie-Hellman key exchange without there being any purpose in the exchange. A large number of near simultaneous such ClientHellos could overload a Server in the absence of rate limiting.

The cookie exchange is illustrated below. In this case, the Server responds to a ClientHello with a HelloRetryRequest and includes a cookie containing some opaque data. The Server then discards the ClientHello.

The RFC recommends that the cookie contains a hash of the ClientHello. When it receives a ClientHello with a cookie, the Server can then validate the cookie without having to retain state. Only if a ClientHello contains a valid cookie and is otherwise identical to the preceding ClientHello (i.e a hash of the ClientHello matches the cookie value) does the handshake protocol continue normally.

```
Client                                           Server

ClientHello
+ key_share              -------->
                                          HelloRetryRequest
                         <--------              + cookie
ClientHello
+ cookie
+ key_share              -------->
                                                 ServerHello
                                                + key_share
                                         {EncryptedExtensions}
                                         {CertificateRequest*}
                                               {Certificate*}
                                         {CertificateVerify*}
                                                  {Finished}
                         <--------        [Application Data*]
{Certificate*}
{CertificateVerify*}
{Finished}               -------->
[Application Data]       <------->        [Application Data]
```
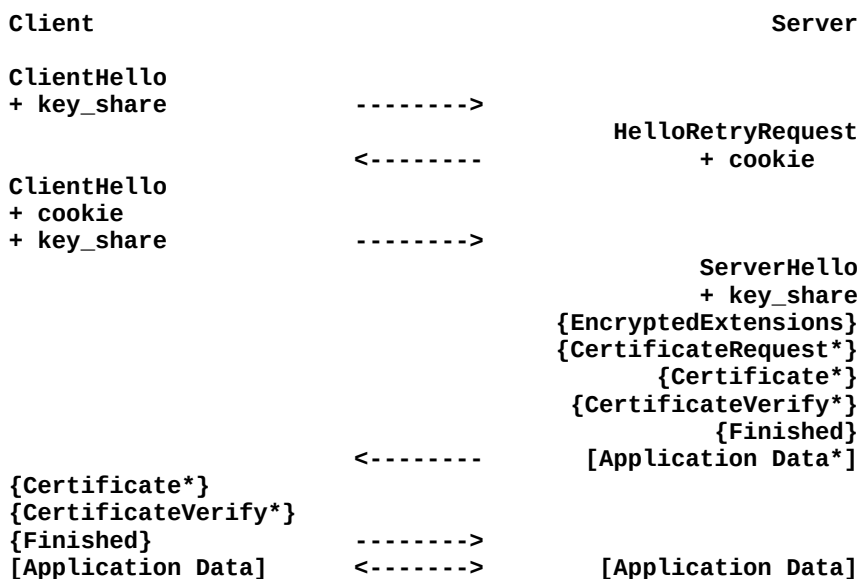
A cookie exchange is a useful way to avoid one type of DoS attack. However, the cost is an extra message exchange for each TLS session.

A.1.7. Pre-shared Keys and Session Resumption

```
Client                                              Server

ClientHello
+ key_share              -------->
                                                 ServerHello
                                                + key_share
                                        {EncryptedExtensions}
                                        {CertificateRequest*}
                                               {Certificate*}
                                         {CertificateVerify*}
                                                   {Finished}
                         <--------      [Application Data*]
{Certificate*}
{CertificateVerify*}
{Finished}               -------->
                         <--------         [NewSessionTicket]
[Application Data]       <------->         [Application Data]
```
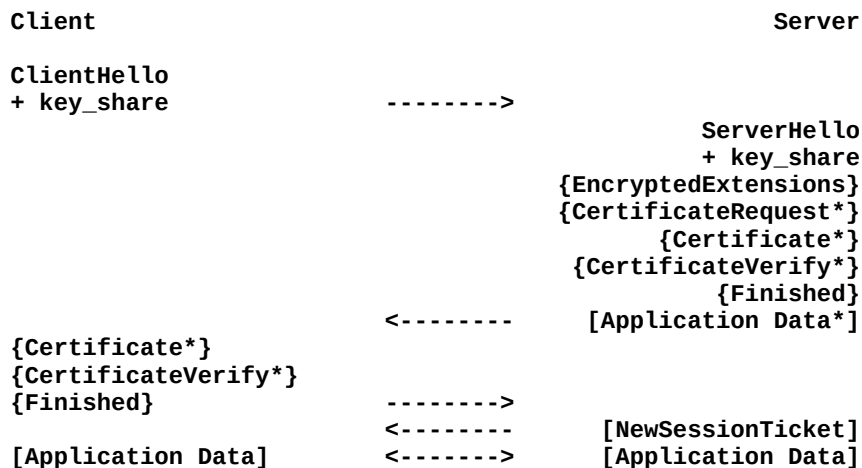
Normally, the TLS Handshake protocol performs both key agreement and authentication. The above shows the characteristic three way handshake, but also includes a Server generated NewSessionTicket message, in this case, sent immediately after the conclusion of the handshake – although this message can be sent at any time afterwards.

The NewSessionTicket message creates a unique association between the ticket value given in the message and a secret PSK derived from the resumption master secret – itself derived from the shared secret determined during key exchange.

The ticket value does not itself need to be protected from an attacker. It is just some opaque identifier that both Client and Server can associate with the secret PSK. It is of no value to an attacker if they do not also know the secret PSK.

This ticket may be used to resume a lost or closed session, or to establish a parallel session- in each case, without the overhead of key exchange and authentication. This is illustrated below.

```
Client                                              Server

ClientHello
+ key_share*
+ pre_shared_key         -------->
                                                 ServerHello
                                            + pre_shared_key
                                                + key_share*
                                        {EncryptedExtensions}
                                                   {Finished}
                         <--------      [Application Data*]
{Finished}               -------->
[Application Data]       <------->         [Application Data]
```
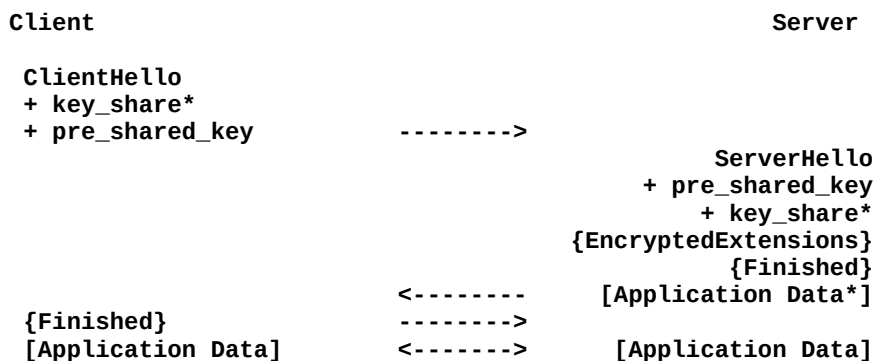
Here, the Client provides the *pre_shared_key* extension containing one or more ticket values provided previously by the Server. In its ServerHello, the Server identifies the accepted ticket also using a *pre_shared_key* extension. As long as both Client and Server use the same Secret PSK a new TLS session can start leveraging the Key exchange and authentication from the session used to send the corresponding NewTicketMessage.

Note that there is no need to exchange certificates or the CertificateVerify message given that authentication is implicit in the possession of the PSK. Application data exchange can also take place immediately after the Server Response is sent.

It is also possible for the Client and Server to perform a new Key Exchange on session resumption in order to ensure "perfect forward secrecy". They include the *key_share* extension in their ClientHello and ServerHello respectively in order to do this. This incurs the overhead of Key Establishment, but avoids the authentication overhead.

A secret PSK is not a one time code and may be used more than once e.g. to establish multiple parallel sessions. For example, a Web Browser may use parallel sessions to obtain images and other resources concurrent to the loading of a web page.

The industry standard does not mandate any specific lifetime for the ticket value. In practice, most implementations will discard unused tickets after a timeout which could be anything from a few minutes to several hours.

A.1.8. Out-of-Band Pre-shared Keys

It is also possible to provision Clients and Servers with pre-shared keys (i.e. ticket values) generated using some other mechanism. For example, an administrator could configure each with a Ticket Value and corresponding secret PSK. This can avoid the need for authentication using X.509 certificates entirely.

The RFC places no strong limits on how out-of-band pre-shared keys are generated. Possible examples include:

- A Client/Server pair agrees a Ticket Value using the NewSessionTicket message as above. The Client provides both the Ticket Value and the secret PSK to another (trusted) Client which then proceeds to use the Ticket Value to establish a secure session with the same server but without the need for Server Authentication.

  Note: If Client Authentication is also performed then the above allows the second client to masquerade as the first and is probably undesirable.

- When both peers are combined TLS Clients/Servers and perform mutual authentication, a TLS session established in one direction may be used to exchange a NewSessionTicket message as above. The exchanged Ticket Value could be used to establish a new session in the reverse direction.

A.1.9. Zero Round Trip Resumption (0-RTT)

0-RTT is a variation of the use of pre-shared keys and is illustrated below:

```
      Client                                        Server

      ClientHello
      + early_data
      + key_share*
      + psk_key_exchange_modes
      + pre_shared_key
      (Application Data*)      -------->
                                                   ServerHello
                                              + pre_shared_key
                                                   + key_share*
                                           {EncryptedExtensions}
                                                   + early_data*
                                                      {Finished}
                              <--------      [Application Data*]
      (EndOfEarlyData)
      {Finished}              -------->
      [Application Data]      <------->         [Application Data]
```
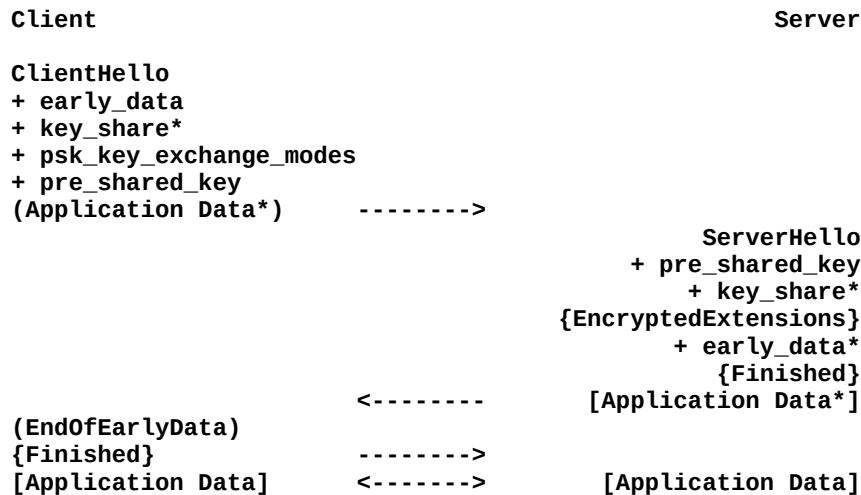
The exchange is for a normal session resumption but includes an *early_data* extension in both ClientHello and ServerHello followed by Application Data on the Client side before it has send the Finished message.

The *early_data* extension signals the intent to send application data before the handshake has completed. The extension also identifies the maximum early data that the Client expects to send. So that the Server can distinguish early data from normal application data (they use different keys), an (empty) EndOfEarlyData Message is sent by the Client prior to completing the handshake.

As indicated in the RFC, with early data there is no opportunity for perfect forward secrecy given that any encryption keys have to be derived directly from the secret PSK. There are also no guarantees that early data cannot be replayed in a subsequent session established using the same Ticket Value.

A.1.10. Application Data Transfer

TLS provides a reliable data stream service that appears the same as that provided directly by TCP. The Application Data Stream is broken up into blocks of an appropriate length, then formatted into a Plain Text Block before always being transformed into Cipher Text prior to transmission.

On receipt, Cipher Text Blocks are decoded and any application data extracted before being appended to the incoming data stream.

A.1.11. Re-Keying

The KeyUpdate (indicating update_requested) message is sent (protected using the current keys) to tell the peer that the sender is updating its key to the next generation of keys.

On receipt, the receiver updates their keys to the next generation and responds with its own KeyUpdate (indicating update_not_requested). Data sequence is assumed (as is provided by TCP) and all messages sent after a KeyUpdate message use the next generation of keys.

Note that it is possible for KeyUpdate (update_requested) messages can cross in flight. Use of the above procedure means that, in this case, the session advances two generation of keys.

A.1.12. Heartbeat

RFC 6520 introduces the Heartbeat exchange for both TLS and DTLS (see A.2). This is used for Dead Peer Detection and may be necessary when the underlying communication service does not itself support this. TCP has an optional keepalive mechanism and if this is in use then the Heartbeat extension is unnecessary.

An Heartbeat Hello extension is defined to allow the use of Heartbeat to be negotiated by a ClientHello/ServerHello exchange. Heartbeat is enabled when both include the extension and permit the other side to send Heartbeat messages.

A Heartbeat Request message may be sent at any time during a TLS Session. The other side must respond with a Heartbeat Response message.

Heartbeat may be used to perform Path MTU discovery and/or a liveness check.

In the former case, dummy user data may be carried in the request and is reflected back in the response.

In the latter case, there is no agreed Heartbeat rate timer and the interval between successive Heartbeat exchange is determined independently by each side. Note that neither side is required to send Heartbeat requests, even when use of Heartbeat has been negotiated.

A.1.13. Session End

An AlertMessage is used to close a TLS Session. If no explicit closure message is sent (e.g. TCP connection release implicitly ends a session) then the TLS Session is vulnerable to a "truncation attack".

In most cases, session closure is performed by each side sending an AlertMessage indicating *close_notify*. No more data is sent after this message. The underlying connection is assumed to support graceful release and hence once a *close_notify* has been sent and received, either or both sides may safely release the connection.

An AlertMessage indicating *user_canceled* may be sent during the handshake phase to indicate user initiated termination prior to handshake completion – and hence prior to application data exchange.

A.1.14. Error Reporting

Other errors are signalled using an AlertMessage indicating the reported error. RFC 8446 lists the many possible errors that can be reported.

Once an error reporting AlertMessage has been sent or received, the underlying connection is released.

A.1.15. Backwards Compatibility Considerations

RFC 8446 provides for backwards compatibility between TLS 1.3 and earlier versions e.g. support of the change_cipher_spec record.

A.2. The Datagram Transport Layer Security (DTLS) Protocol

RFC 9147 describes a datagram version of the TLS and is intended for use with a Datagram Transport Service such as that offered by UDP. The protocol messages and session structure are as near as possible the same as for TLS. Where there are differences is consequential on the use of an unreliable datagram transport as opposed to a reliable stream mode connection.

As a result, DTLS must overcome:

- Packet Loss: Retransmission timeouts and acknowledgements have to be added.

- Re-ordering: Messages can be re-ordered as a normal consequence of datagram transport. Packet sequence numbers are needed to overcome this.

- Fragmentation: TLS assumes that the underlying transport can stream very large messages with loss of data or re-sequencing. DTLS must operate in an environment where each datagram has a limited size. Large messages (e.g. those containing certificates) will have to be fragmented into multiple datagrams and re-assembled on arrival.

- Replay Threats: packet duplication may occur maliciously or by accidental duplication in the network. DTLS must be able to detect and reject duplicates.

4.5.1 DTLS Record Structure

In order to support datagram communication, DTLS defines both epoch and sequence numbers:

- An Epoch Number is an unsigned 64 bit integer that is incremented on each Key Update.

- A Sequence Number is an unsigned 64 bit integer that is incremented for each Plain Text record and set to zero at the start of each epoch.

The DTLS Record Structure changes from TLS. In particular, the Plain Text header now includes:

- The low order 48 bits of the 64-bit unsigned sequence number and

- The least significant 16 bits of the 64 bit epoch number that is incremented on each Key Update.

- An optional 16 bit length – omission implies that the packet fills the remainder of the datagram

The Cipher Text header is also revised to include:

- An 8 or 16 bit sequence number (an encrypted and truncated version of the Plain Text sequence number)

- The least 2 significant bits of the epoch number and,

- A variable length connection ID (see RFC 9146).

Epoch numbers allow for the possible re-ordering of Key Update and Application Data messages. Even after a Key Update has been received, an out-of-order Application Data message may still be received from an earlier epoch. The epoch number allows for it to be identified and associated with the correct key generation.

DTLS messages may be combined into the same datagram. Due to datagram size limits, it may also be necessary to fragment a DTLS Message over more than one datagram.

A.2.1. The Handshake Protocol

Handshake messages are large and are vulnerable to packet loss and mis-ordering. They may also require fragmentation and re-assembly. Large Handshake Messages e.g. the Certificate Message may need to be fragmented. Any such fragmentation and re-assembly is part of the Handshake Protocol rather than the underlying DTLS Record Layer.

DTLS Handshake messages can thus include *fragment_offset* and *fragment_length* indications which may be used to both indicate that fragmentation has taken place and used for message re-assembly. These indications are part of the Handshake message header, as updated for DTLS.

DTLS Handshake messages must also be subject to retransmission timeouts to protect against loss of all or part of a message. There is no partial re-transmission capability and hence lost of a fragment results in re-transmission of the entire message.

When small enough, Handshake messages can be concatenated in to the same datagram.

The EndOfEarlyData message is not carried forward into DTLS. Its function is replaced by the use of epochs.

4.5.1.1 The initial Handshake

A general model for the DTLS Handshake protocol is shown below and copied from RFC 9147. Note the more explicit grouping into "Flights" indicating a set of messages sent without interruption using more or more datagrams.

```
Client                                          Server

                                            +--------+
  ClientHello                               | Flight |
                    -------->               +--------+

                                            +--------+
                    <--------   HelloRetryRequest | Flight |
                                    + cookie  +--------+


                                            +--------+
  ClientHello                               | Flight |
  + cookie          -------->               +--------+


                                 ServerHello
                          {EncryptedExtensions}  +--------+
                          {CertificateRequest*}  | Flight |
                                 {Certificate*}  +--------+
                              {CertificateVerify*}
                                     {Finished}
                    <--------   [Application Data*]


  {Certificate*}                            +--------+
  {CertificateVerify*}                      | Flight |
  {Finished}        -------->               +--------+
  [Application Data]
                                            +--------+
                    <--------        [ACK]  | Flight |
                                [Application Data*]  +--------+

  [Application Data] <------->     [Application Data]
```
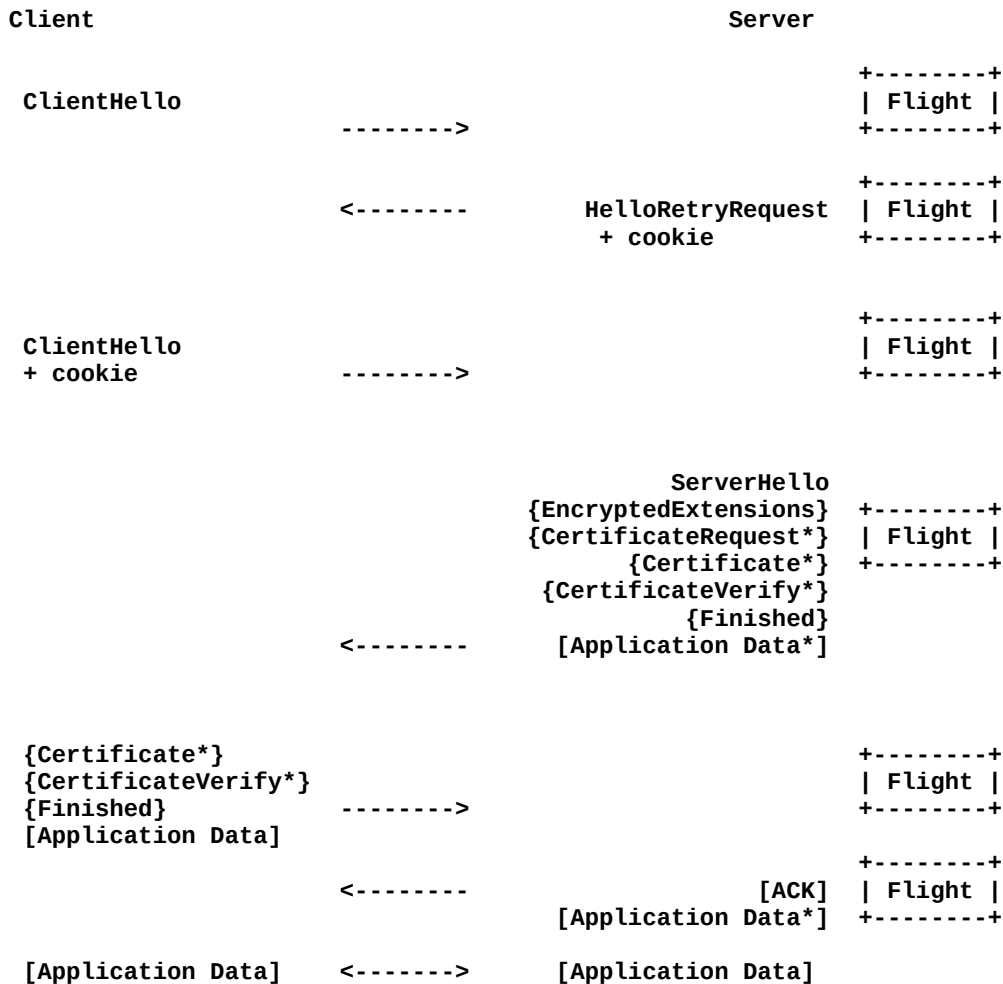
The above assumes that the optional cookie exchange is being used. If either the initial ClientHello or the HelloRetryResponse is lost, then retransmission of the ClientHello is sufficient to recover from this loss. The Server is stateless after the transmission of the HelloRetryRequest and will simply process the re-transmitted ClientHello as a new message.

If the ClientHello + cookie or the ServerHello is lost then retransmission of the ClientHello + cookie is sufficient to recover from the loss. The Server is stateful at this point and will reply to a ClientHello retransmission by retransmitting the ServerHello and the remainder of the "Flight".

Note that this is the same as the case where there is no cookie exchange. i.e. retransmission of a ClientHello (on its own) should cause the Server to retransmit the ServerHello Flight. However, the cookie is arguably an important Denial of Service countermeasure for datagram communication and is recommended for DTLS.

The next Flight sent by the Client (including its Finished message) may be lost. A new ACK message is introduced to cope with this case. If the ACK is not received within a timeout, the Client retransmits the message Flight and the Server will again respond with the ACK and any application data sent with the same Flight.

A.2.1.1. Session Resumption

The Handshake protocol for PSK based session authentication and resumption is shown below and as copied from RFC 9147.

```
Client                                          Server

 ClientHello
  + early_data
  + psk_key_exchange_modes                      +--------+
  + key_share*                                  | Flight |
  + pre_shared_key                              +--------+
 (Application Data*)      -------->

                                          ServerHello
                                       + pre_shared_key
                                          + key_share*  +--------+
                                    {EncryptedExtensions} | Flight |
                                             {Finished}  +--------+
                         <-------      [Application Data*]


                                                      +--------+
 {Finished}              -------->                    | Flight |
 [Application Data*]                                  +--------+


                                                      +--------+
                         <-------                [ACK]  | Flight |
                                    [Application Data*]  +--------+

 [Application Data]      <------->      [Application Data]
```
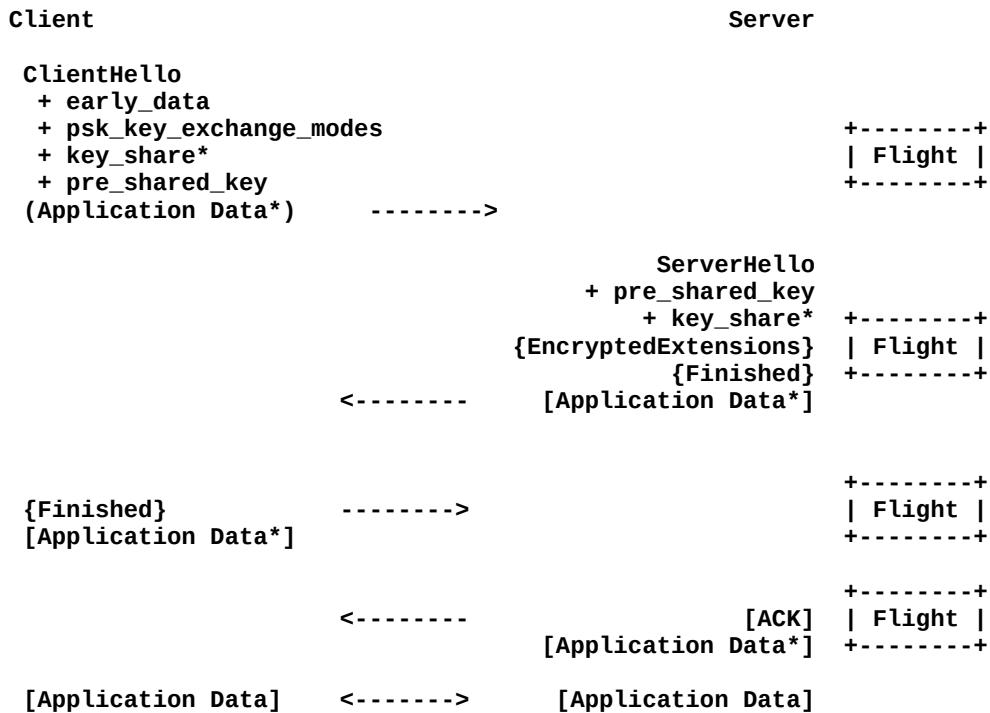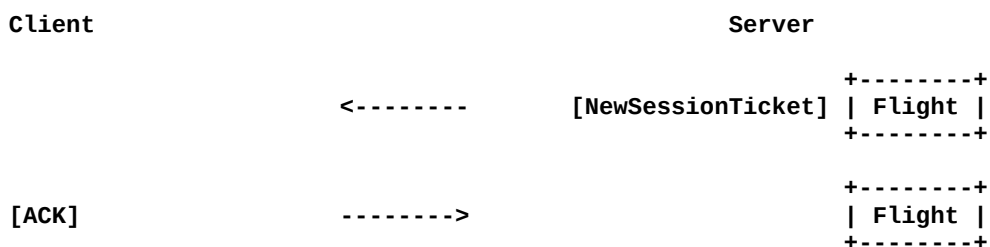
The above is largely as would be expected. The main change from the TLS is the introduction of the ACK message which, as above, is used to allow the Client to detect loss of its Finished Message.

A.2.1.2. New Session Ticket Message

In TLS, the message is unacknowledged given the assumption of a reliable stream based transport. In DTLS, the ACK Message is also used to confirm receipt of a NewSessionTicket message and to force re-transmission if the Server does not receive an ACK within the timeout.

This is illustrated below:

```
Client                                          Server

                                                      +--------+
                         <-------      [NewSessionTicket] | Flight |
                                                      +--------+

                                                      +--------+
 [ACK]                   -------->                    | Flight |
                                                      +--------+
```

A.2.1.3. ClientHello Races

If a DTLS endpoint is able to act as both a Server and a Client and a ClientHello is sent both from a well known port on the Client to a well known port on the Server then there is a risk of a ClientHello race when both sides attempt to start a DTLS session near simultaneously.

In this situation there needs to be an unambiguous winner. It is not possible to have two simultaneous sessions between the same pair of ports. A common tie breaker rule needs to be implemented on both sides – such as lowest source IP Address in numerical order wins.

On the other hand, a simple way of avoiding the problem is for Clients to always use ephemeral ports for their local port.

A.2.2. Application Data

The DTLS user may be assumed to be aware that they are using a datagram transport and that datagrams can be lost, delayed or duplicated. The DTLS user should also be aware of the maximum datagram size and to not exceed this.

Assuming the above:

- There is no need for DTLS to recover from Application Data loss or delay. This is a user problem.

- Duplication could be also be seen as a user problem. However, DTLS should prevent replay attacks and uses record sequence numbers to detect and discard duplicates. A sequence number window will also reject all "old" datagrams that are outside of that window.

- Application Data that exceeds the maximum datagram size may be rejected. Fragmentation and re-assembly of application data is not supported by DTLS.

Computation of the maximum datagram size will depend upon the record protection algorithm. It is computed by subtracting from the maximum UDP datagram size:

- The size of the Plain Text Header

- Any inflation of the Cipher Text over the Plain Text e.g. due to padding, HMACs or a side effect of the encryption algorithm.

- The size of the CipherText Header.

DTLS will reduce the maximum datagram size by at least 16 bytes (assuming no traffic padding).

A.2.3. Alert Messages

Alert Messages (error and session closure) are not retransmitted. Instead, DTLS specifies that any messages received with an epoch/Sequence number later than the alert message must be ignored.

This can result in "half closed" sessions where one side is unaware that the other has initiated closure or reported an error. In this case, it should be assumed that any Application Data messages sent are "going down a black hole". The user application should be able to recognise this case and take appropriate action.

A.2.3.1. Session Closure

The "half closed" session risk can be countered by placing a timeout on idle sessions i.e. if no datagrams have been received of any content type on a given session for longer than the timeout then the session is automatically considered closed without needing to send any Alter Messages.

This timeout need not be identical on both Client and Server.

A.2.4. Heartbeat

With DTLS, Heartbeat may be used to:

- Perform path MTU discovery in situations where the maximum datagram size is unknown and hence to determine empirically the maximum datagram size.

- Perform Dead Peer Detection

- Counter the half-open session threat.

However, Heartbeat imposes an overhead due to the need to exchange Heartbeat messages and this extra overhead may be undesirable for slow speed networks.

A.2.5. Summary

DTLS is very similar to TLS and cannot be understood without first understanding TLS.

The Record Layer is updated to provide for epoch/sequence numbers and which are necessary to manage key update and for replay protection.

The Handshake Protocol additionally supports fragmentation and re-assembly of large messages, and introduces a new ACK message to ensure that message loss is recoverable in all situations.

The User Application must be aware of the nature of datagram transport and must respect the maximum datagram size, and the risk of data loss and re-ordering. Half-closed sessions are also a risk that the application must be able to counter.

Appendix B.   Security Concepts

B.1. Communications Security

The purpose of Communications Security is to protect network assets from unauthorised use and to protect network users from a deliberate attack. The set of threats to networks and their users include:

- Masquerade, where an unauthorised user impersonates another authorised user and thereby gains access to information and other assets that that user owns or has access to.

- Modification, where data is modified while being transferred between two end users in order to cause an unauthorised action to be invoked.

- Replay, where a copy of valid data that was transferred earlier is re-sent by an attacker in order to cause an unauthorised action to be invoked.

- Eavesdropping, where unauthorised access to data takes place.

- Denial of Service, where an attacker prevents authorised access to communication services.

By no means all applications are vulnerable to each of the threats listed above. For example, Eavesdropping Threats are not relevant to applications that deal in data that is otherwise in the public domain.

Where the threat is relevant then an appropriate security mechanism is needed to counter the threat. These can include both cryptographic and physical security mechanisms.

Cryptographic mechanisms can protect data from modification, replay and eavesdropping. They can also be used to authenticate users to each other and hence be used to protect against masquerade.

Physical Security mechanisms are needed to protect cryptographic mechanisms (e.g. by protecting secrets used by the cryptographic mechanism) and to prevent Denial of Service through physical disruption of assets. Wireless communications also permits Denial of Service through "jamming" and physical security mechanisms may hence be needed to monitor, locate and close down jamming signals.

Denial of Service attacks can also be made against cryptographic mechanisms, especially those which are computationally intensive, by inducing unnecessary computational effort as a result of receiving a message from an unauthorised party. The design and implementation of cryptographic algorithms needs to take this into account, and ensure that such attacks are sufficiently mitigated.

Cryptographic mechanisms make use of two basic techniques: secret key cryptography and public key cryptography. These are supported by other concepts including Message Hashes, keyed Hash Message Authentication Codes (HMACs), Digital Signatures and Key Management Schemes.

B.2. Secret Key Cryptography

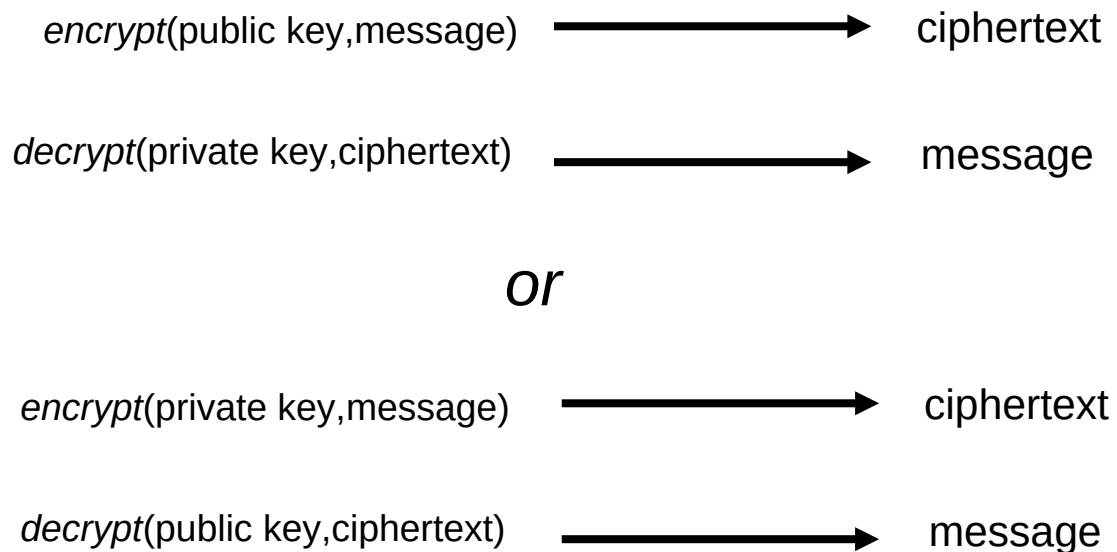*encrypt*(key,message) ⟶ ciphertext

decrypt(key,ciphertext) ⟶ message

Secret key cryptography uses an algorithm that transforms data into an apparently random bit stream, but which can also transform it back again into the original data stream without loss of information. The algorithm takes a single parameter (the key) and data so encrypted with a given key, can only be decrypted with the same key. As long as there are (a) a very large number of possible key combinations and (b) major differences in the encrypted stream for only small differences in the key or unencrypted data, a secret key encryption algorithm can protect data from eavesdropping.

The main weakness is that both sender and receiver must use the same key and key distribution is thus itself a potential vulnerability – as, should an attacker intercept a key when it is distributed, the attacker then has access to all communications protected by the key. Key lifetime is also an issue as the more data that is encrypted using a given key, the easier it becomes for crypto-analysis to break the encryption.

The strength of a secret key system is that it is based on a shared secret (the key). When the algorithm is properly constructed, the only realistic means of breaking the encryption is trying all possible values of the key. By choosing a sufficiently large number of key combinations this can be made computationally infeasible.

Examples of Secret Key encryption schemes include the Defense Encryption Standard (DES), an improved version known as triple-DES (3DES), the Advanced Encryption Standard (AES), Blowfish and IDEA.

B.3. Public Key Cryptography

encrypt(public key,message)   ➜   ciphertext

decrypt(private key,ciphertext)   ➜   message

or

encrypt(private key,message)   ➜   ciphertext

decrypt(public key,ciphertext)   ➜   message

Public key cryptography similarly uses an algorithm that transforms data into an apparently random bit stream, but which can also transform it back again into the original data stream without loss of information. The difference from secret key encryption is that two mathematically related keys are used instead of a single secret key; the keys are created in pairs – one key is designated as the private key while the other becomes the public key. Each user is given its own unique key pair, and when one is used to encrypt the data, the data can only be decrypted using the other key.

As the name suggests, the public key can be well known and published in directories, whilst the private key must be kept secret by its user.

Public key cryptography can be used to encrypt data but is typically computationally much slower than a secret key algorithm. Public key cryptography is typically used for authentication through digital signatures.

Public key cryptography has two main weaknesses:

- the public and private keys are mathematically related and there is a consequential risk that a private key could be derived from the public key, and

- the risk that an attacker could replace a genuine user's public key (e.g. in a directory) and hence masquerade as that genuine user (see B.7.3).

Public key algorithms are based on mathematically "hard" problems. For example, RSA is probably the best known public key algorithm, and its hard problem is the identification of the prime number factors of very large numbers[2].

---

[2] Formally, the RSA Problem is more general defined, but integer factorisation is regarded as the most likely way the RSA Problem can be solved.

Ensuring that a private key cannot easily be derived from the public is a major design issue for public key cryptography algorithms and public key algorithms, such as RSA, can only remain in use[3] as long as there are no known means for deriving a private key from a public key in a short enough time to make this useful.

With RSA, a key length of at least 2048 bits is recommended[4] to make this computationally feasible, with some experts recommending 4096 bits in order to take account of technology development. However, 2048 bit is believed acceptable for normal use, taking into account that there is no long term need to protect data.

B.4. Message Hashes

hash(message) ⟶ digest

Hash algorithms compute a fixed-length digital representation (known as a message digest) of an input data sequence (the message) of any length. They are called "secure" when it is computationally infeasible to:

1. find a message that corresponds to a given message digest, or

2. find two different messages that produce the same message digest.

Any change to a message will then, with a very high probability, result in a different message digest.

Examples of Message Hash functions include MD5 [RFC1321], and the Secure Hash Algorithm (SHA) group of Hash Functions [RFC4634].

The size of the message digest can vary between the algorithms. For example, SHA-1 produces a 160-bit message digest, while MD5 produces a 128-bit hash value.

On their own, hash functions offer no protection against an attacker as they can be easily regenerated and replaced after a message has been modified. However, they are useful in protecting against accidental modification and can be used to support digital signatures (see B.6.3) and HMACS.

Both MD5 and SHA-1 are now regarded as "not good enough" and the more powerful SHA-256 is recommended for use with authentication algorithms.

B.5. Keyed Hash Message Authentication Codes (HMACs)

khash(key,message) ⟶ HMAC

Keyed Hash Message Authentication Codes (HMACs) have been developed to create a class of message digest algorithms that provide a proof of both integrity and message origin.

---

[3] Because no public key algorithm can be guaranteed unbreakable forever, this class of algorithm is not viewed as suitable for applications where data protection is required in the long term rather than simply "on the day of transfer".

[4]This number increases in response to available computational power and may be out-of-date.

[RFC2104] provides a commonly used algorithm for the use of message hashes to support authentication. This is achieved by including a shared secret within the scope of the message digest, such that the message digest can only be created and verified by users that possess the same shared secret.

The [RFC2104] algorithm applies a message hash such as MD5 or SHA-1 twice. Once to the message text prefixed by a shared secret key that has been XORed with a well known bit string, and secondly by applying the message hash function to the result of the first hash again prefixed by the shared secret key, now XORed with a different well known bit string. The result is then truncated. The RFC recommends to a maximum of 80 bits (from the 128 bits of MD5 or the 160 bits of SHA-1).

Truncation has advantages (less information on the hash result available to an attacker) and disadvantages (less bits to predict for the attacker).

[RFC2104] is not the only HMAC scheme available. HMACs based directly on block cipher (secret key) algorithms such as DES and AES are also possible.

[APP-CRYPTO] also casts doubt on whether the security of an HMAC based on message hashes is as secure as those based on block ciphers, noting that "implicit but unverified assumptions are often made about the properties that [unkeyed hash functions] have".

B.6. Authentication

HMACs used a shared secret to perform authentication. Other forms of authentication are also possible, such as passwords and Digital Signatures.

B.6.1. Pre-shared Keys

When a secret key algorithm is used for encryption, the simplest way of deploying it is to provide all parties to the communication with a pre-computed key as a shared secret. This is shared information by all parties and is hence known as a Pre-Shared Key (PSK). With PSK, in anything other than a very small network, the overhead of distributing the key securely, rapidly becomes excessive, especially as effective security requires that keys are changed regularly.

PSK is thus usually deprecated for any wide-scale deployment. However, this does not stop its continued use. For example, in WiFi the relatively weak Wireless Encryption Protocol (WEP) uses PSK, as does WPA2.

B.6.2. Passwords

Passwords are a very familiar albeit a weak form of identification and are another variation on the shared secret approach. Passwords can be either exchanged directly or used as a key to encrypt or provide an HMAC for same random text, which can then be verified using the password as the key.

In general, a password is a pre-shared key in text form.

B.6.3. Digital Signatures

encrypt(Private Key,hash(message)) ⟶ signature

An HMAC can provide authentication but not repudiation. This is because an HMAC can be generated by both the sender or the receiver of a message, using the single shared key. It authenticates a message to either party to the communication but not to third parties; there being no additional proof as to whether the sender or receiver generated the HMAC.

Digital Signatures are designed to provide non-repudiable authentication of a message's source. For this reason, they are based on Public Key Cryptography rather than a shared secret. That is, the private key is used to sign the message and the public key to verify the signature. Because the private key is kept secret by its owner, successfully verification using a public key proves the assertion that the owner of the private key generated the message's digital signature.

Several examples of Digital Signature algorithms exist. One popular approach is to generate a message digest, for example using SHA-256, and to then encrypt the result using signer's private key. The signature can then be verified by:

1. Regenerating the SHA-256 Message Digest

2. Decrypting the Digital Signature using the purported signer's public key

3. Comparing the two results. If they are the same then the signature is verified. Otherwise, verification fails.

Other common Digital Signature Schemes include the Digital Signature Algorithm (DSA) [DSA].

B.7. Key Agreement Schemes

Digital Signatures can be computationally expensive and the number of time a given key should be used should anyway be kept to a minimum. If a single public/private key pair was used to provide continuing authentication throughout a single dialogue, not only would this cost in terms of CPU time, but it would drastically shorten the key lifetime and hence put up the cost of key management.

Even with a shared secret, it is undesirable to use it too often as this gives more opportunities for an attacker to determine the shared secret.

Instead, it is usually preferred (e.g. RFC5996 - IKEv2) to use a Digital Signature or a Pre-shared Key to perform initial authentication and to use another mechanism to generate dynamic session keys.

Session keys are ephemeral and are generated at the start of a secure session, rather than distributed prior to communication.

B.7.1. Diffie-Hellman Key Agreement

A common means to generate session keys is the *Diffie-Helman* key agreement scheme. A basic implementation of this scheme – for key agreement between "Alice" and "Bob" can be described as follows:

- Alice and Bob agree to use a prime number p and base g.

- Alice chooses a secret integer a, then sends Bob the value of $g^a \bmod p$

- Bob chooses a secret integer b, then sends Alice the value of $g^b \bmod p$

- Alice computes $K = (g^b \bmod p)^a \bmod p$

- Bob computes $K = (g^a \bmod p)^b \bmod p$

They have both arrived at the same value K, because $g^{ab} = g^{ba}$. This relies on the fact that exponentiation is commutative in modular arithmetic.

Neither a nor b have to be exchanged and remain secret. This is for as long as it is difficult to compute a from $g^a \bmod p$ even when prime number p and base g are well known. Provided that a, b and p are large numbers, this remains true. This is based on a mathematically hard problem known as the *discrete logarithm problem*.

In a typical implementation, a and b are chosen randomly, and then discarded to ensure *perfect forward secrecy* (see B.7.2), The base g is taken from a cyclic group G, and is assumed to be known to attackers. The *Diffie-Hellman* exchange does not provide authentication and the exchange needs, for example, to be signed using a digital signature if *Alice* and *Bob* need to authenticate each other.

[RFC2631] specifies another and commonly used way of using *Diffie-Hellman* key agreement making direct use of a public/private key pair, where K is generated from[5]:

$K = g \wedge (X_a * X_b) \bmod p$

Where X_a is the private key of Alice and X_b is the private key of Bob.

As it is highly undesirable to communicate private keys, Alice and Bob actually compute:

$K = (Y_b \wedge X_a) \bmod p = (Y_a \wedge X_b) \bmod p$

Where Y_a is the public key of Alice and Y_b is the public key of Bob.

Assuming that: the private and public keys are related by:

$Y = (g \wedge X) \bmod p$

This can be shown to work through the normal rules of algebra.

[RFC2631] goes on to categorise different modes of use. In the static mode of use, the public/private key pair is statically assigned to either or both parties, while in the ephemeral

---

[5] ^ is used to denote exponentiation and * to denote multiplication.

mode, the public/private key pair is generated immediately before the exchange takes place.

The RFC also describes a way to generate multiple keys of various lengths from a single *Diffie-Hellman* exchange. This is achieved by successively generating a message hash from the concatenation of the generated *K* and the ASN.1 DER encoding of a Key Info value that includes a sequence number that changes each time the hash is generated. This key generation is performed to avoid the high computational cost of generating *K*. and for each session key needed.
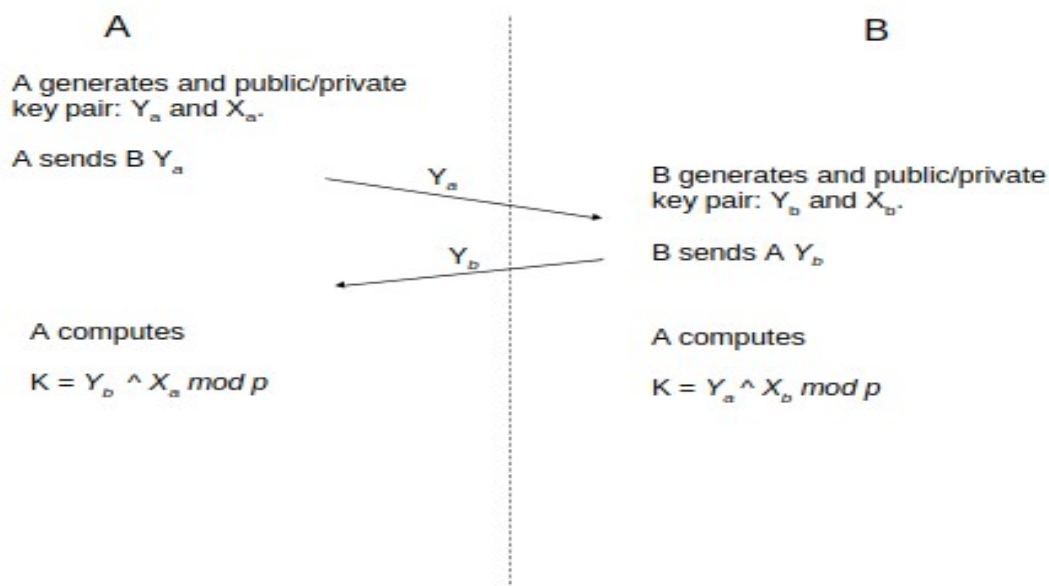

Figure 3-1 Diffie-Hellman Key Agreement

Diffie-Hellman is a very practical key agreement scheme. However, as noted, computation of *Diffie-Hellman* exponentials can take significant computing power and this gives rise to a possible Denial of Service attack. This is because *Diffie-Hellman* key agreement will often take place at the same time or even before authentication has taken place and, by repeatedly attempting *Diffie-Hellman* exchanges with a remote system, an attacker can tie up excessive computing resource even though authentication will always fail.

Rate limiting is a typical mitigation strategy to counter this problem.

B.7.2. Perfect Forward Secrecy

Diffie-Hellman key agreement can also be used to give what is known as Perfect Forward Secrecy. This occurs when a new (ephemeral) public/private key pair is generated for each new *K* required, and this key pair and all session keys are discarded without record at or before the end of the session. There is then no way for an attacker to decrypt the session even if the static public/private key pair used to authenticate the session are later compromised.

For example, IKEv2 specifies the use of ephemeral public/private key pairs, but allows local policy to determine how often a new key pair is generated.

B.7.3. Trusting a Public Key

As noted in B.3, a major vulnerability of a public key scheme is that an attacker could replace *Alice's* public key with their own and thus masquerade as *Alice* to *Bob*. Several schemes have been developed to allow trust in the binding of a public key to its correct owner, the most important of which is the concept of Digital Certificates supported by a Public Key Infrastructure.

B.7.4. Public/Private Key Pairs in the Secure Shell

The popular Secure Shell (SSH) is a network protocol that allows data to be exchanged over a secure channel between two computers. Encryption provides confidentiality and integrity of data. SSH uses public-key cryptography to authenticate the remote computer and allows the remote computer to authenticate the user, if necessary.

SSH has a simple and effective approach to key management. When using SSH, each SSH Client and Server generates their own Public/Private key pair. So that they may be given permission to log on to a particular Server, a Client must provide, using a secure channel, a copy of their Public Key to the Server's Administrator who then adds it to the list of approved clients. Clients can then log on to the Server with their identity authenticated by encrypting a randomly generated piece of data using their private key. The Server can validate this by decrypting it using their public key.

Under SSH, Clients are not usually given a copy of the Server's Public Key. Instead, they are provided with its "Public Key Fingerprint". This is a (usually shorter) message digest taken over the public key and is more human readable. The first time an SSH Client connects to a Server, it displays the fingerprint of the public key offered by the Server. The human user is then expected to compare the reported fingerprint with that expected and to confirm it. From then on, as long as the Server's public key remains unchanged, the SSH Client will happily connect to it.

The use of the Public Key Fingerprint is intended, in this case, to prevent "man in the middle attacks". More generally, it is a useful technique where a human user is required to validate the authenticity of a public key or a similar complex or large data structure.

SSH exhibits two useful techniques:

- the provision of a public key via a secure channel – the secure channel provides trust in the key, and

- the use of a "digital fingerprint" to provide a proof that the public key is the expected public key – allowing a human user to check the key against some other published piece of information.

B.7.5. PGP "Web of Trust"

The Pretty Good Privacy (PGP) package is a popular means of securing Email. It also uses Public Key Cryptography and again, each user is expected to generate their own Public/Private Key Pair.

In PGP, public keys are exchanged informally often by direct physical exchange. Correspondents may also pass on keys to others and that way keys are learnt through a series of trust relationships. When loaded into a user's PGP "keyring", the public key

fingerprint may also be used as an additional piece of corroborating information when accepting the validity of the key.

B.7.6. Public Key Certificates

The SSH approach to public key distribution works well in small deployments where long lived relationships exist and a relatively high setup up cost (in terms of human overhead) is acceptable. The PGP "Web of Trust" works well in informal networks of friends and colleagues. However, neither can really be said to provide an efficient and robust scheme for wide-spread use of public keys. The concept of Public Key Certificates has been developed to enable an efficient and robust key management scheme for the wide scale deployment of Public Key Cryptography.

Public Key Certificates exist to allow public keys to be widely published and exchanged through insecure channels, whilst still providing a means to validate that a given public key belongs to a given user and not to an attacker masquerading as that user.

ITU-T Recommendation X.509 is the reference standard for Public Key Certificates, and was developed in order to provide a mechanism for publishing public keys as part of digitally signed "certificates". Each certificate includes the identity of its owner and the public key, and is digitally signed by a trusted third party – the Certification Authority (CA). A public key can then be authenticated by validating the CA's digital signature on the certificate.

In order to do this, the CA's public key must be in the possession of the authenticator. The CA's public key is typically distributed in a self-signed CA certificate. Being self-signed this proves integrity but not authenticity. The CA certificate must still be distributed using a secure channel. However, once it has been distributed securely, the CA certificate can be used to validate all certificates that it signs both before and after it is distributed.

For example, in Windows, a list of trusted root certificates is distributed using Windows Update; in Linux, the CA certificates are distributed by the Linux Distro as system updates.

Digital certificates depend on a secure channel to provide initial trust in a single public key and the leverage this trust relationship to all parties to the CA's "web of trust".

X.509 certificates are now in widespread use in applications such as secure website access and online banking.

The procedures, policies, protocols and algorithms used by the CA together form a Public Key Infrastructure (PKI) and a PKI must be specified before Public Key Cryptography and Digital Certificates can be used to provide authentication.

B.8. Public Key Infrastructures

In cryptography, a public key infrastructure (PKI) is an arrangement that provides for trusted third party vetting of, and vouching for, user identities. It also allows binding of public keys to users. This is usually carried out by software at a central location together with other coordinated software at distributed locations. The public keys are typically distributed in Digital Certificates.

ITU-T recommendation X.509 [X509] provides the most commonly used certificate format. [RFC2527] provides guidance on the Certificate Policy and Certification Practices statements for certification authorities and public key infrastructures.

An example of a widely used and hence public PKI is provided for website authentication and was originally set up by Netscape. In this PKI, there are a number of recognised and trusted Certificate Authorities and which sign the certificates used to authenticate each website. The self-signed root certificate for each such CA is distributed and updated by the operating system provider.

B.8.1. The Certification Authority

In order to operate a PKI, one or more "root" Certification Authorities (CAs) must be established. The establishment of a CA is outside of the scope of this paper. However, the typical procedures of operation are described here.

The CA is responsible for:

1. Creating its own self-signed digital certificate and distributing this securely to all users of the PKI. The distribution method must be secure enough for the users to have confidence that when given the CA's self-signed certificate, it is the genuine CA certificate.

2. Receiving Certificate Signing Requests from users.

3. Authenticating a Certificate Signing Request and the requested purpose.

4. Generating a signed certificate (signed by the CA) and returning it to the requesting user.

5. Maintaining and distributing lists of revoked certificates.

Subordinate CAs may also be established by various Stakeholders, such as National Administrations, International Organisations and Internet Service Providers, etc.. In this case, the CA's certificate is signed by the root CA to which they are subordination.

As long as each user of the PKI has access to and trusts the authenticity of the self-signed certificate of each root CA. There is no need for the root CAs to have any formal relationship. Alternatively, the root CAs may cross-sign certificates for each other, which can be used to authenticate user certificates when only a subset of the root CA certificates are available.

B.8.2. The Digital Certificate

The management and distribution of Digital Certificates is a primary purpose of the PKI and the CA. A common format for the certificate is that specified by ITU-T Recommendation X.509, as profiled by [RFC3280]. This comprises information identifying the user or system that is the subject of the certificate, optionally including their Name, Postal Address, Organisation and contact details, the user's Public Key, and other attributes, such as those which identify the purposes for which the certificate may be used. The certificate is digitally signed by an encrypted message hash generated using the private key of the CA.

When a certificate is signed, it is given a time limited validity period. It may no longer by used once it has expired. During this validity period, a CA may revoke an issued certificate by publishing it in a "Certificate Revocation List", which is distributed regularly to all users of the CA.

A format for Digital Certificate distribution is specified in [RFC2315].

```
XCertificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = GB, ST = London, L = Westminster, O = Snake Oil (Sales) Ltd,
OU = Certification Services Division, CN = Snake Oil Root CA, emailAddress =
cobra@badguys.com
        Validity
            Not Before: Feb 28 14:34:07 2024 GMT
            Not After : Nov 24 14:34:07 2026 GMT
        Subject: C = GB, ST = London, L = Westminster, O = Snake Oil (Trading)
Ltd, OU = Secure Web Server, CN = localhost, emailAddress = python@badguys.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:de:c2:7d:b2:d8:d1:fc:3a:d3:fc:9f:9c:20:dd:
                    5c:48:f3:eb:db:57:b8:0c:56:00:42:37:2f:ce:9a:
                    20:d5:31:19:48:c3:f5:b9:db:86:f6:21:c2:17:c2:
                    5a:36:71:9e:f2:32:c7:85:d1:36:a5:3f:c1:76:14:
                    f0:bf:1d:52:8b:04:88:8d:ea:db:c6:dd:1e:43:71:
                    db:68:30:7f:9f:41:c3:1c:62:d9:ae:ad:a8:59:3f:
                    25:c9:68:4a:72:97:9e:aa:0a:df:24:5d:72:23:1b:
                    5d:4f:5a:74:cc:55:5e:df:91:d4:29:28:9f:65:96:
                    2b:47:75:46:cb:0f:75:a3:ae:1e:f0:dc:d4:42:ec:
                    73:bc:1b:89:67:f2:2e:34:fa:39:8e:79:94:d0:47:
                    f6:36:dc:9f:af:a8:d5:69:df:4b:8c:20:ca:4c:0e:
                    39:4b:4c:cc:3f:c3:99:af:b2:f3:40:5c:92:f3:fb:
                    93:b9:ea:61:5d:b7:c9:77:85:a0:4e:1c:c5:fc:49:
                    cd:a9:da:ec:2c:2f:64:ee:8e:3d:d1:2d:26:dc:06:
                    77:5c:a5:d0:b5:a2:70:5b:0a:bc:f3:53:78:19:50:
                    c7:59:17:8f:85:93:ab:96:0c:98:72:af:90:16:42:
                    72:78:bb:0a:15:8e:6f:2c:f4:1d:4b:e5:e4:5c:67:
                    2e:15
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                49:AF:87:7A:BB:01:A7:42:E6:17:76:4A:7F:7C:20:DD:FC:F4:61:B8
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication,
Microsoft Server Gated Crypto, Netscape Server Gated Crypto
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                E2:BD:0D:C4:17:71:71:B9:A5:C8:D1:18:C0:61:16:1A:4D:70:80:34
    Signature Algorithm: sha256WithRSAEncryption
```

```
      Signature Value:
          2e:ee:0d:22:80:ea:9c:2e:c7:34:ef:e9:6f:c4:b8:b7:5b:d2:
          0c:e1:ae:1b:d2:e9:18:bd:e5:85:be:37:bc:bf:f7:e7:6e:9a:
          ed:fc:d4:7a:8d:18:1c:e5:a7:ce:4c:db:01:ac:ad:67:a0:1b:
          d7:bd:03:be:4c:41:66:5e:8d:ad:83:c5:3d:90:b9:f9:1c:65:
          25:59:5a:12:03:65:16:5a:a7:01:6f:ec:d7:a3:98:57:a8:d5:
          c4:ad:20:dc:ef:03:87:9e:ff:61:ba:1b:56:42:1c:91:5a:b5:
          18:59:a4:82:a2:42:c1:60:f7:70:d8:12:b6:05:49:1f:9e:2b:
          f5:61:00:b7:7a:34:08:08:2d:24:be:76:6a:1a:cc:53:92:54:
          66:8a:7b:5e:8f:c4:94:22:72:c9:4c:e4:4c:ec:12:ea:9b:47:
          4a:4a:8f:2e:c3:32:6c:f2:69:a9:2c:42:67:d4:93:be:40:20:
          53:9d:c9:76:ad:3f:57:e7:13:e1:57:7b:1a:44:e5:40:d0:7b:
          e7:c6:dd:f8:1e:0b:0e:9f:56:f6:d6:26:9b:fe:14:78:df:d0:
          6a:84:50:05:d8:2f:c8:df:6a:05:d3:a0:af:72:2c:6b:1c:0a:
          47:83:17:62:da:91:87:9c:43:3a:a7:3d:2a:b2:b1:79:ab:85:
          2f:49:f9:3a:a1:66:ca:b9:00:c4:f9:6c:e5:9b:c0:89:7f:2e:
          a4:ed:ff:6a:ce:6d:99:6a:22:85:3d:09:2b:3b:21:fd:e6:08:
          cf:60:ca:8b:06:91:59:14:0a:06:36:3b:a8:e8:21:75:12:11:
          80:5a:3a:f2:50:79:06:14:d2:f0:af:fa:69:8b:38:c2:45:9a:
          b7:e5:d8:fc:bf:db:a3:c1:a9:62:3e:ab:04:45:09:60:fa:a3:
          e8:e7:91:4a:69:2c:fc:ea:a5:68:b1:b2:31:c5:d5:c1:e8:70:
          6b:9d:43:ad:f5:d5:8f:b0:ba:00:34:85:8d:d9:fe:29:a9:4b:
          fc:1c:90:63:69:10:59:f8:13:47:9b:af:b0:a2:67:fd:63:90:
          0f:74:f8:84:63:f0:5a:71:ec:f9:fe:69:9a:25:d0:95:20:b9:
          d7:17:df:71:75:69:db:b4:87:46:86:33:32:d9:eb:3f:44:30:
          ed:66:0d:af:1b:e7:65:d1:1d:2e:f3:32:b1:0a:a7:20:c0:37:
          16:f1:5f:cc:ba:74:8f:86:c7:68:d3:c9:3f:be:cc:9b:50:16:
          9d:9f:3d:19:24:5f:a1:93:9e:2b:17:ab:aa:a1:ae:bc:44:dd:
          6b:e9:71:64:c2:8f:3c:39:fa:28:fb:56:89:96:6c:45:b0:79:
          22:a5:38:9d:44:a2:db:35
```

Figure 5-2 Example X.509 Certificate (Text Formatted)

B.8.3. The Self-Signed Certificate

The purpose of the self-signed certificate is to provide a means of distributing the public key of the CA.

In order to create a self-signed certificate, the CA must first generate a new Public/Private Key pair. This is the key pair it will use for certificate signing.

The CA creates a self-signed certificate by assembling an X.509 certificate using the newly generated Public Key, its own identity and includes an indication that the certificate is a CA certificate. The certificate is then digitally signed with the corresponding Private Key.

The self-signed certificate must be securely distributed to all of the users.

B.8.4. Certificate Signing Requests

Each user of the PKI that requires a certificate, must first generate a new Public/Private Key pair. This becomes the user's public/private key pair.

The user then creates a Certificate Signing Request (CSR). This comprises the Public Key generated above, the identity of the requesting user and a digital signature generated using the user's private key.

The Certificate Signing Request is then sent to the CA.

[RFC2986] defines a format for the certificate signing request.

B.8.5.   Validating and Signing the Certificate Request

The CA must satisfy themselves that the CSR's digital signature is valid and that the request has come from the user identified in the CSR and that they are authorised to request a certificate.

If these checks are passed, then the CA will create an X.509 certificate for the requestor, using the information contained in the CSR. The CA may indicate limited purposes for the certificate by appropriate setting of the KeyUsage and ExtendedKeyUsage fields of the certificate. The certificate must also be given a time limited validity period.

The CA will sign the certificate using its private key and return the certificate to the requesting user.

B.8.6.   Certificate Validation

The CA's users may assert their identity by signing some text using their private key and offering the signed certificate as a proof of their identity. The receiving user validates the certificate using the CA's public key obtained from the self-signed CA certificate that they had previously received from the CA over a secure channel. Once the certificate has been validated, the public key it contains can be used to validate the digital signature on the text received from the other user. If this succeeds then a proof is given that the sender was the owner of the certificate.

This proof in turn depends on the validating user trusting the CA to have correctly authenticated the original CSR and that the CA certificate is genuine.

B.8.7.   Certification Revocation

From time to time, the CA may need to cancel (revoke) a certificate before its validity period expires. This may, for example, be because its owner has informed the CA that the private key has been compromised and hence the user needs to replace the Public/Private Key pair and be issued a new certificate.

A certificate is revoked by adding to a list of revoked certificates - the Certificate Revocation List (CRL). This list comprises all the certificates that have been revoked but are still within their validity date and is regularly distributed to all users of the CA.

The CRL may also be distributed in RFC 2315 format, It is digitally signed by the CA and itself has a validity date. It may also be empty. Each of the CA's users should normally expect to have an up-to-date copy of the CRL.

B.8.8.   Sub-ordinate Certification Authorities

Although only a single CA is required, for administrative reasons, it may be desirable to delegate CA functionality to various Stakeholders.

A Stakeholder may set up its own CA and issue certificates for its users as described above. So that these certificates can be recognised by other users, the CA must:

- Create a CSR for its Public/Private Key Pair and submit this to the root CA.

- The root CA should validate the request and return an X.509 certificate to the subordinate CA, signed using the root CA's private key. The X.509 also contains a BasicConstraints field that indicates that the certificate owner is itself also a CA.

This certificate is then returned to the subordinate CA.

Instead of using a self-signed certificate, the subordinate CA uses the certificate issued by the root CA. It distributes this certificate to its users along with the self-signed certificate for the root CA.

When a user of the subordinate CA presents its X.509 certificate to another user, it must also include the issuing subordinate CA certificate, unless it knows otherwise that the other user will already have the subordinate CA certificate. The receiving user may then validate the subordinate CA certificate (using its copy of the root CA certificate) and then the user certificate in turn.

The subordinate CA certificate and the root CA certificate are said to for a certificate chain. There is no formal limit to the length of such a chain.

B.9. The Implementation of a Certification Authority

B.9.1. Functional Requirements

A Certification Authority operates offline and does not have to implement any communications protocols. In practice, a CA needs to be able to:

1. Receive Certificate Signing Requests (CSRs) from users. Many existing CAs allow CSRs to be pasted into web pages and submitted to the CA with the web page.

2. Validate the CSR. This is often the most difficult problem for a CA as the CSR could have just have easily come from an attacker as from a genuine source. An independent means of validation is required, such as a telephone call to a known contact at the organisation submitting the CSR, to ensure that it came from the claimed source.

   A CA may, for example, require that a formal registration takes place before CSRs can be submitted, with an organisation required to offer several proofs that they are an appropriate source of CSRs and to provide contact points for independent verification of CSRs.

3. Sign the certificate, including a unique serial number which can be used later, for example in certificate revocation. This requires a suitable software tool (e.g. openssl).

4. Save the details of the certificate, including the requesting organisation and the serial number in a database. This is so that it can be recalled, if necessary for revocation.

5. Return the signed certificate to the requester. The certificate is intended to be made public and is only directly useful to the holder of the

corresponding private key. It thus does not need to be returned via a secure channel. Email to a registered contact point should be sufficient.

B.9.2. Software Requirements

The Open Source *OpenSSL* [OPENSSL] package is a well known and complete implementation of the toolset required for a small CA and its users, using X.509 certificates, RSA or DSA public/private keys and the certificate formats referenced above. This includes the means to generate public/private key pair, create a CSR and self-signed CA certificates, and to for a Ca tosign CSRs. It also includes a simple file system oriented database.

OpenSSL is available for both Linux and Windows.

A larger CA will require a better database than that provided with OpenSSL. If the CA is web based, then this be readily achieved using PHP and MySQL. The PHP web scripting language has built-in interface to both OpenSSL and MySQL and can thus be used to build the system required to support a full scale CA.