eZ80® Family of Microprocessors

ZTP Network Security SSL Plug-In

User Manual

UM020107-1211

This publication is subject to replacement by a later edition. To determine whether a later edition exists or to request copies of publications, visit www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Revision History

Each instance in the Revision History table below reflects a change to this document from its previous version. For more details, click the appropriate links in the table.

Date	Revision Level	Description	Page Number
Dec 2011	07	Updated for the SSL v2.4.0 release.	All
Feb 2009	06	Updated for the SSL v2.3.0 release.	All
Nov 2008	05	Updated for the SSL v2.2.0 release; updated Figure 4, Figure 5, and Linking the SSL Libraries sections.	14, 15, 19
Aug 2007	04	Updated for the SSL v2.1.0.	All
Jul 2006	03	Global changes: updated SSL and ZTP version numbers.	All
Apr 2006	02	Global changes: updated SSL and ZTP version numbers; removed all content related to ZTP v1.3.4.	All
Jan 2006	01	Formatted to current publication standards.	All

Table of Contents

Introduction

Zilog's TCP/IP Network Security SSL Plug-In provides security for TCP connections established between a client and a server using the Zilog TCP/IP Software Suite (ZTP). The SSL handshake protocol is used to authenticate the server and arrive at a shared secret between the client and server that is used to encrypt all application data transferred over the SSL session. Additionally, each transferred message contains a message authentication code that can detect if its data has been altered during transition.

Features

This package is only compatible with the ZTP Software Suite of the same version. For example, the ZTP Network Security Plug-In package `X.Y.Z` is only compatible with ZTP Software Suite `X.Y.Z`; it is not compatible with ZTP Software Suite `X.Y+1.Z` or `X.Y.Z-1`.[1]

The ZTP Network Security SSL Plug-In includes support for the following versions of the SSL handshake protocol:

- SSL version 2 (SSLv2) client and server support
- SSL version 3 (SSLv3) client and server support
- TLS version 1 (TLSv1) client and server support

Each version of the SSL protocol can operate in any of the following modes, independent of other SSL handshake protocols:

- Client Only mode
- Server Only mode
- Client Server mode

You can run one, two or all three versions of the SSL handshake protocol simultaneously.

The ZTP Network Security SSL Plug-In provides support for the following cryptographic functions.

- Supported public key algorithms:
  – RSA
  – DSS (using the DSA)
  – DH

1. Throughout this document, `X.Y.Z` refers to the ZTP version number in `Major.Minor.Revision` format

- Supported digest algorithms:
  - MD5
  - SHA1
  - Keyed MD5 (HMAC_MD5)
  - Keyed SHA1 (HMAC_SHA1)

- Supported symmetric cipher algorithms:
  - RC4 (128-bit)
  - DES (56-bit)
  - Triple DES (3DES, 168-bit)
  - AES (128-bit or 256-bit)

Limitations of the ZTP Network Security SSL Plug-In

The ZTP Network Security SSL Plug-In does not support the following SSL features:

- Client authentication
- Anonymous Diffie-Hellman cipher suites
- Support for the MD2 digest algorithm
- Support for the IDEA or RC2 symmetric ciphers
- Support for the Fortezza key exchange algorithm

Architecture

Figure 1 displays the main software modules of that comprise the ZTP Network Security SSL Plug-In.

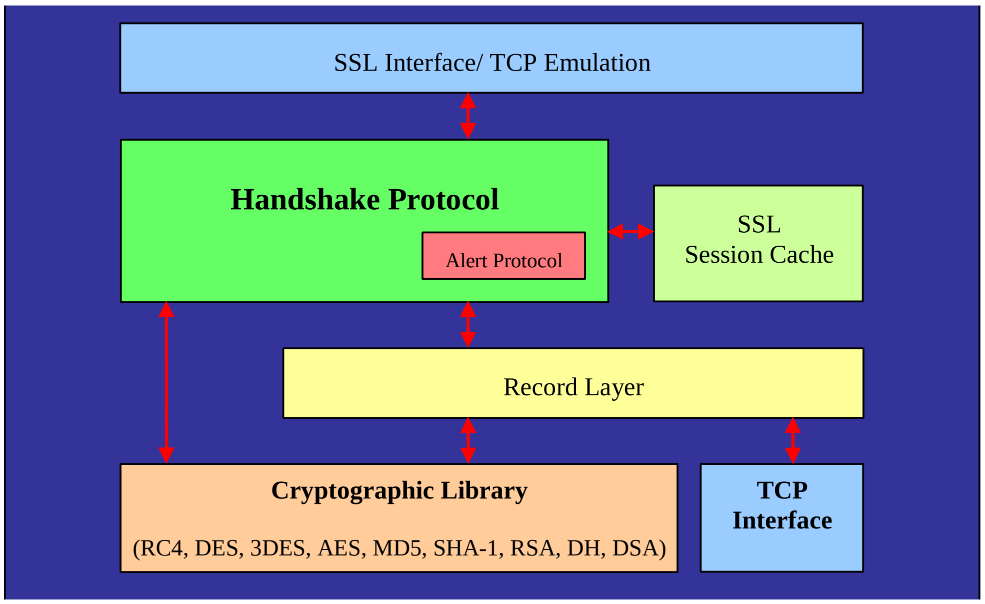


Figure 1. Software Modules in the ZTP Network Security SSL Plug-In

Each of the following SSL modules is described in this section.

- TCP Interface Module
- SSL Record Layer Module
- SSL Session Cache Module
- SSL Interface Module
- SSL Cryptographic Module

TCP Interface Module. This module uses the ZTP TCP API to establish TCP connections and exchange SSL data. It also uses the stream sockets interface (open, bind, close, read, write).

SSL Record Layer Module. SSL Record Layer module is above the TCP Interface module. This module is responsible for framing all SSL handshake messages and application data. After an SSL session is established, the record layer will pad all messages to a multiple of the cipher’s block size, compute a message authentication code on the data, fragment the message, and then encrypt each of the fragments. Upon receiving the message, the record layer reassembles inbound fragments, decrypts the message, verifies the message authentication code, and passes the message to the handshake protocol for additional processing. For application-level data, the record layer allows the data to be received from the upper SSL interface.

The handshake protocol module is responsible for establishing SSL sessions. This module actually contains six sub-modules:

1. SSLv2 Client
2. SSLv2 Server
3. SSLv3 Client
4. SSLv3 Server
5. TLSv1 Client
6. TLSv1 Server

The module used to establish an SSL session depends on the configuration of the ZTP Network Security SSL Plug-In. It is possible for multiple handshake modules to be active at the same time.

SSL Session Cache Module. Adjacent to the handshake module is the SSL session cache module, which is used to store information about the established SSL sessions. If the same client and server attempt to establish another session in the future, the session cache can be enabled to reduce the number of handshake messages that must be exchanged, which will in turn reduce the session establishment time. This reduction is primarily a result of not having to perform complex public key algorithms.

SSL Interface Module. Above the handshake module is the SSL interface module. This layer exposes the SSL API to upper-layer applications. Other than the SSL-specific initialization commands (`Initialize_SSL`, `SSL2_ClientInit`, `SSL2_ServerInit`, `SSL3_ClientInit`, `SSL3_ServerInit`, `TLS1_ClientInit`, and `TLS1_ServerInit`), this interface exposes the same TCP interface as used by the TCP Interface module. This exposure allows user applications that are written to use ZTP's TCP interface to be seamlessly ported to use SSL.

SSL Cryptographic Module. The final module in the ZTP Network Security SSL Plug-In is the cryptographic module, which contains the digest algorithms, ciphers and public key algorithms used by the SSL protocol to secure application data.

How to Use SSL

The `Initialize_SSL` API must be called to enable the SSL Interface layer used by applications to securely transfer data. This API must be called only one time during system initialization, regardless of how many SSL client and server tasks are created. Additionally, an initialization call must be made for each version of the SSL handshake protocol that will be supported by the application. This initialization routine is accomplished by calling one or more of the following APIs:

- `SSL2_ClientInit`
- `SSL2_ServerInit`

- `SSL3_ClientInit`
- `SSL3_ServerInit`
- `TLS1_ClientInit`
- `TLs1_ServerInit`

Client mode support is enabled by calling the corresponding `xxx_ClientInit` API. Server mode support is enabled by calling the corresponding `xxx_ServerInit` API. Client-Server mode is enabled by calling `xxx_ClientInit` and `xxx_ServerInit` API. An optional HTTPS server can also be created by calling the `https_init` API.

The code fragment that follows shows an example of each of these initialization steps.

```
/*
* Initialize the SSL Layer
*/
Initialize_SSL();

/*
* Initialize each handshake protocol for client
* and server support. Each protocol is configured
* to use the same certificate chain. Ephemeral
* Diffie-Hellman parameters are used for SSLv3
* and TLSv1.
*/
   SSL2_ClientInit();
   SSL2_ServerInit( &CertChain, NULLPTR );
   SSL3_ClientInit();
   SSL3_ServerInit( &CertChain, &DheParams );
   TLS1_ClientInit();
   TLS1_ServerInit( &CertChain, &DheParams );

   /*
   * Launch the HTTPS server over SSL
   */

https_init(http_defmethods,httpdefheaders,website,443);
```

After the initialization steps are complete, the application programs set up SSL sessions and securely transfer data using an API that is almost identical to that of the TCP API running on the underlying ZTP system.

TCP-based applications in ZTP use the `open`, `bind`, `send`, and `receive` API sockets to establish TCP connections and transfer data. To use SSL, ZTP applications still use the same API. The only difference is the use of the `SOCK_SSL` socket type instead of the `SOCK_STREAM` socket type.

This user manual explains these concepts and offers a considerable amount of information related to SSL configuration files. Careful modification of these configuration files will alter the default behavior of the ZTP Network Security SSL Plug-In.

Difference Between SSL Versions

This section offers a brief summary of the differences between the multiple versions of the SSL protocols supported by and relevant to the ZTP Network Security SSL Plug-In. This material is not intended to be an explanation of the SSL handshake protocols.

SSL Version 2

SSL version 2 is the oldest and simplest of the SSL handshake protocols. The default set of *cipher suites* defined in the SSLv2 specification (known as *cipher specs* in SSLv2) use RSA for the key exchange algorithm and MD5 as the digest algorithm. The default set of ciphers supported in the SSLv2 specification are:

- RC2
- RC4
- IDEA
- DES
- 3DES

Note: The ZTP Network Security SSL Plug-In does not support RC2 or IDEA.

One potential security flaw of the SSLv2 protocol is that it is susceptible to man-in-the middle types of attacks in which an attacker can trick the actual client and server into using a relatively insecure cipher suite. This situation is possible because the SSLv2 client has the final choice of SSLv1 cipher suite used during the session. This choice is usually based on the set of mutually-supported cipher suites that the SSLv2 server returns in its *hello* message.

However, these SSLv2 handshake messages are not protected; therefore, it is possible that an attacker could intercept the server's *hello* message and modify the list of mutually-supported ciphers so that just a single weak cipher remains. This intercept can trick the client to use a weaker cipher suite than it would have ordinarily chosen based on the original message received from the server. The attacker then tries to determine the weak cipher's symmetric key to gain access to the encrypted data.

To overcome this problem, the SSLv3 and TLSv1 protocols maintain a running digest of all SSL handshake messages used to establish a session. After the session is established, the client and server both encrypt the digest and send it to the other side for verification. If this verification step fails, the session is not established. Therefore, if an attacker modifies one of the SSLv3 or TLSv1 handshake messages, the SSL session will not be established.

SSLv3 and TLSv1 also expand the set of public key algorithms used to establish an SSL session; both allow the use of the DSA and DH algorithms. SSLv3 also supports the Fortezza key exchange algorithm, although this particular algorithm was later dropped from the TLSv1 protocol; it is not supported by the ZTP Network Security SSL Plug-In.

SSLv3 and TLSv1 use MD5 and SHA1 for computing message authentication codes. Therefore, security flaws in either of these algorithms cannot be exploited to gain access to the secure data. Also, these protocols use different keys in the computation of message authentication codes and data encryption. In contrast, SSLv2 uses the same key to compute the message authentication code and encrypt the data. Therefore, it is easier for an attacker to gain access to secure data using SSLv2 because a successful attack on either a cipher or a digest algorithm will compromise this secure data.

The main difference between SSLv3 and TLSv1is that TLSv1 uses a complex pseudorandom function generator based on keyed MD5 and SHA1 digests (`HMAC_MD5` and `HMAC_SHA1`) when selecting random values required by the TLSv1 handshake protocol. The PRF function must digest thousands of bytes of data to produce a few dozen output bytes. This amount of processing can have the effect of scrambling the data into excellent pseudorandom values, yet it does so at the expense of additional computations and slower overall operation.

In general, the SSLv2 protocol is less secure than the SSLv3 or TLSv1 protocols. However, the additional computations performed in SSLv3 and TLSv1 protocols to secure the session causes the session establishment times of these protocols to be longer than for SSLv2. In addition, because of the complexity of the TLS pseudorandom function generator, it takes longer to establish TLSv1 sessions than it does to establish SSLv3 sessions.

SSL Handshake Protocols

This chapter presents an overview of the SSL handshake protocols and some background information about security concepts.

Before SSL begins transferring encrypted application data, an SSL session is established between the SSL client and the SSL server. The establishment of a session is initiated by the SSL client.

The following processes occur during this session establishment:

1. The client verifies the identity of the server. This verification is performed by analyzing a certificate that the server sends to the client when a new session is established. The SSL protocol optionally allows the server to request a certificate from the client so that the server can verify the client's identity. However, the SSL server in ZTP does not implement client authentication.
2. The client and server decide on a set of cryptographic algorithms to be used to exchange a secret key, encrypt/decrypt data (cipher), and ensure message integrity (through a one-way hash function). The combination of a key exchange algorithm, a cipher algorithm and a hash algorithm is called a *cipher suite*.
3. The client generates a secret value, called a Master Key, that is used to derive additional keys for encrypting/decrypting data exchanged between a client and a server. This key is sent to the server using the selected key exchange algorithm; and is protected using the information in the server's certificate (the server's public key) and other SSL handshake messages.
4. Because the key exchange algorithm is asymmetric, only the server that possesses the corresponding private key recovers the Master Key generated by the client.
5. The client and server independently generate read and write keys from the Master Key; these keys are used to encrypt/decrypt data with the cipher algorithm.
6. The client and server exchange test messages to ensure both sides are using the correct Read and Write keys. The test message is composed of data exchanged using the handshake protocol. In the case of TLSv1 and SSLv3, the test message is a hash of all handshake messages used to establish the SSL session. This message is also encrypted using the negotiated cipher suite. If each party is able to decrypt the message and verify its contents then both parties are using the same symmetric key. This also proves that the server is in possession of the private key corresponding to the public key in the server's certificate and completes the authentication of the server.

When a session is successfully established, every byte of data exchanged between the client and server is packaged into an SSL data record. Each data record contains a field called the message authentication code (MAC), which is computed using the Hash function defined for a particular cipher suite used. The entire record is then encrypted and sent to the peer. The peer decrypts the inbound message, verifies the MAC code; and if found acceptable, it presents the data to the upper layer application. When all of the required information is exchanged between the client and server, the underlying TCP connection is severed.

Security Concepts

This section introduces some basic aspects of security as related to SSL. This information is not intended to be a security reference or to explain the SSL protocol.

Identity. Identity is a set of attributes that uniquely distinguishes one particular entity from other similar entities. Before the SSL client can establish a session, it must identify the SSL server with which it must communicate. The identity of the server typically consists of a host name (or IP address) and an underlying TCP port number.

Authentication. Authentication is the process of validating an entity's identity. Upon establishing an SSL session handshake, the SSL server sends the client an X.509 certificate that the client uses to verify the identity of the server. One of the fields in the certificate is a digital signature created by a third party (or possibly the server itself) called the certificate issuer. By signing the certificate, the issuer vouches for the identity of the server and asserts that there is a binding between the subject of the certificate (the SSL server) and the public key contained in the certificate, implying that the actual SSL server to which the certificate is issued is in possession of the corresponding private key.

After executing a public key exchange algorithm, the client and the server arrive at the same shared secret if the server is in possession of the private key corresponding to the public key in the server's certificate. Therefore, if the client trusts the certificate issuer, then the client can be assured of the server's identity and begin transferring sensitive information.

Notes:

1. The TLSv1 and SSLv3 specifications allow for the use of completely anonymous cipher suites in which neither client nor server authentication is performed. However, the ZTP Network Security SSL Plug-In does not support the use of anonymous cipher suites.
2. A trust relationship can be hierarchical in nature. A client can obtain a certificate from an unknown server that was signed by an issuer that the client does not know/trust. However, if the client obtains an issuer's certificate, and if that certificate is signed by a trusted issuer, then the client can implicitly trust the intermediate certificate issuer and also trust the server's certificate. In the SSLv3 and TLSv1 protocols, the server sends the client a list of certificates. The first certificate contains the SSL server's public key, and any following certificates are the X.509 certificates of the entity that issued the preceding certificate. These chains terminate the self-signed certificate belonging to the root certificate authority.

Cipher. A cipher is an algorithm that transforms plain text into encrypted text, and vice versa. In terms of security, ciphers are used to provide privacy. Even if an encrypted message is intercepted, the plain text content of the message is not visible, and therefore the communication between endpoints is maintained as private.

Cipher algorithms require a special input called a *key* to encrypt/decrypt data. This key is used to uniquely scramble the data as it passes through the cipher algorithm. Cryptographically-strong ciphers are capable of producing very different output blocks for a given plain text block if only a few bits in the key are modified. In general, the longer the key (in bits) the harder it is to determine a plain text message from examining its cipher output.

Ciphers are broadly classified as:

- Symmetric ciphers
- Asymmetric ciphers

Symmetric Ciphers. A symmetric cipher uses the same shared key to encrypt and decrypt a message. Therefore, before a symmetric cipher is used to transfer encrypted data, it is necessary for both parties to possess the same secret key. Figure 2 displays the typical flow of symmetric cipher encryption and decryption.

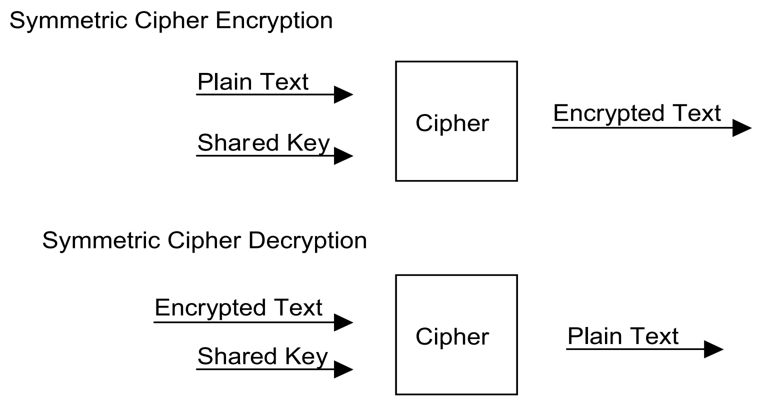


Figure 2. Symmetric Cipher Encryption and Decryption

One of the challenges with symmetric algorithms is to maintain the shared secret as truly secret. For example, if there are 100 clients that communicate with a particular server using a shared secret and this secret is compromised by one of the clients, then all 101 systems must be updated with a new shared secret.

Asymmetric Ciphers. Asymmetric algorithms use different keys to encrypt and decrypt data. Asymmetric algorithms typically use a *public* and *private* key pair. Therefore, unlike symmetric algorithms, it is not necessary to distribute a shared secret to all parties involved before encrypted data transfer occurs. Figure 3 displays the typical flow of asymmetric cipher encryption and decryption.

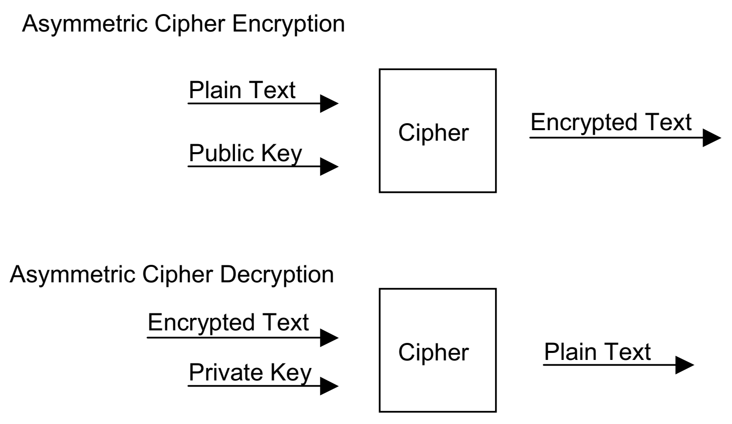


Figure 3. Asymmetric Cipher Encryption and Decryption

In the context of SSL, the server possesses a private key which is not distributed or shared with any client. The corresponding public key is contained in the server's X.509 certificate and freely distributed to prospective clients when they initiate a new SSL session. Therefore, unlike symmetric ciphers, there is no risk associated with a public key being compromised by a client because the public key is not a secret.

The disadvantage of asymmetric ciphers is that they are more computationally intensive than symmetric ciphers. As a result, asymmetric ciphers run much slower than symmetric algorithms. The difference in performance can be a few orders of magnitude, increasing as key strength is increased.

Note: The SSL protocol uses an asymmetric cipher (key exchange algorithm) to exchange the Master Key, and it uses a symmetric cipher to encrypt/decrypt the upper layer data blocks when an SSL session is established.

Stream Cipher. A stream cipher is a symmetric cipher that operates on an arbitrary-sized input message to produce an output message of the same length. The algorithm expands a cryptographic key into a key stream in which its length matches the length of the input text. The input text and key stream are exclusively ORed to produce the final cipher text output message.

Block Cipher. A block cipher is a symmetric cipher that breaks an input message into fixed-sized blocks. Padding may be required to ensure that the input text is an exact multiple of the block size. The block cipher algorithm uses a key to convert the plain text blocks into cipher text blocks on a block-by-block basis.

Hash Function. A hash function takes an arbitrary amount of input data and produces a fixed-sized hash, or digest, of the message. Cryptographic hash functions are one-way functions. It is impossible to determine the original message from a hash of that message. Hashes are commonly used in digital signatures and message authentication codes.

Message Integrity. Prior to encrypting an SSL data record, the SSL protocol computes a one-way hash on the data in the record as well as on the state information pertinent to the SSL session (secret key + message sequence number). The output of the hash function is called a *message authentication code*. If only the originator and intended recipient of the message know the correct state information used to compute the hash, then it is unlikely that an attacker can modify the message in transit without the recipient detecting an error on the MAC.

X.509 Certificate. The SSL protocols require that the server have a certificate that is passed to the client for authentication purposes. The X.509 standard specifies the format of information in the certificate. The certificate contains information such as the identity of the server to which the certificate was issued, a time period over which the certificate is valid, the server's public key, the identity of the certificate issuer, and a digital signature of the certificate generated by the issuer. The signature is created using a hash of the certificate and encrypted using an asymmetric cipher with the issuer's private key.

If a client has the issuer's public key (which can also be in the certificate), then the client can validate the signature and verify the identity of the server. When the server proves that it is in possession of the private key corresponding to the public key in the certificate, the client trusts the server and begins exchanging sensitive data.

The X.509 certificate is specified using a platform independent data modelling language called abstract syntax notation (ASN.1). Encoding of data values in the actual certificate follows ASN.1 distinguished encoding rules (DER format).

Optionally, the SSL protocols allow the server to request a certificate from the client so that it can authenticate the client. However, few clients are likely to have valid certificates, and the server does not request a certificate from the client. The ZTP Network Security SSL Plug-In SSL server does not support client authentication, nor does it request a certificate from the client.

Getting Started

This chapter is a summary of the steps required to run the SSL demo sample project provided with the ZTP Network Security SSL Plug-In. Subsequent chapters provide detailed configuration information. For additional setup information, refer to the ZTP Network Security SSL Plug-In Quick Start Guide (QS0059).

Packages

Table 1 lists both the international and U.S. versions of the ZTP Network Security SSL Plug-In package.

Table 1. ZTP Network Security SSL Plug-In Install Packages

Package Name	Description
SSLX.Y.Z_INT	International version of the ZTP Network Security Plug-In (no source code for the cryptographic functions).
SSLX.Y.Z_US	U.S. version of the ZTP Network Security Plug-In (full source code).

Installation

Prior to installing one of the ZTP Network Security SSL Plug-In packages, you must install the ZTP Software Suite integrated with the ZDS II – eZ80Acclaim! development tools. The install program will place the ZTP Network Security SSL Plug-In in the same folder where the underlying ZTP base package is installed.

The default directory depends on which code version of ZTP has been installed.

Object Code
`C:\Program Files\Zilog\ZTP_x.y.z_Lib_ZDS\ZTP`

Source Code
`C:\Program Files\Zilog\ZTP_x.y.z_Src_ZDS\ZTP`

Note: The `crypto` directory is only included in the U.S. installation of the ZTP Network Security SSL Plug-In, in which `X.Y.Z` refers to the ZTP Software Suite version.

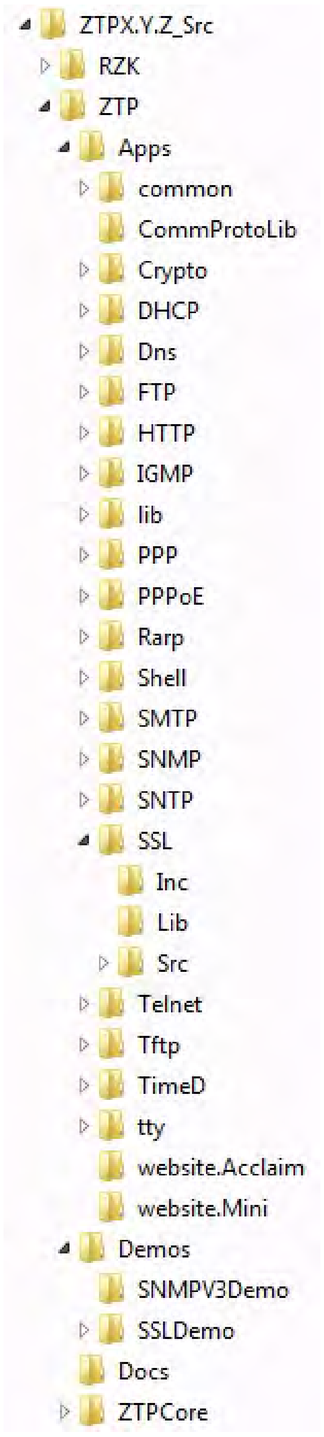
Directory Structure

After installing the ZTP Network Security SSL Plug-In on your PC several new folders and files will be added to the directory in which the underlying ZTP software suite was installed.

Figure 4 displays the directory structure of a ZTP-based system after this plug-in package has been installed. The following new folders were added to the original ZTP installation:

- `Apps\crypto`
- `Apps\SSL`
- `SSLDemo`

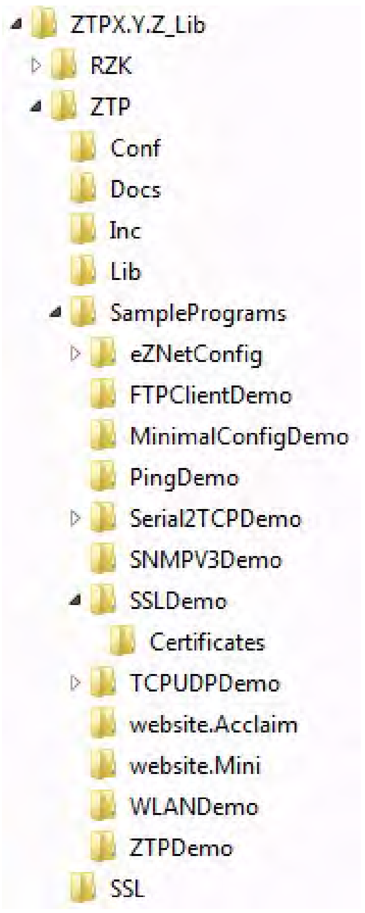
In addition, an `SSL_Crypto` folder is added to the `Inc` folder. SSL-related configuration files are added to the `Conf\Opt` directory.



```
ZTPX.Y.Z_Src
  RZK
  ZTP
    Apps
      common
      CommProtoLib
      Crypto
      DHCP
      Dns
      FTP
      HTTP
      IGMP
      lib
      PPP
      PPPoE
      Rarp
      Shell
      SMTP
      SNMP
      SNTP
      SSL
        Inc
        Lib
        Src
      Telnet
      Tftp
      TimeD
      tty
      website.Acclaim
      website.Mini
    Demos
      SNMPV3Demo
      SSLDemo
    Docs
    ZTPCore
```

Figure 4. Directory Structure for a ZTP-Based Source System

Figure 5 displays the directory structure of a ZTP-based library system after the plug-in package has been installed. The `SSLDemo` folder is added to the original ZTP installation. SSL related configuration files are added to the `\ZTP\Conf\` directory of the ZTP library package.



```
ZTPX.Y.Z_Lib
  RZK
  ZTP
    Conf
    Docs
    Inc
    Lib
    SamplePrograms
      eZNetConfig
      FTPClientDemo
      MinimalConfigDemo
      PingDemo
      Serial2TCPDemo
      SNMPV3Demo
      SSLDemo
        Certificates
      TCPUDPDemo
      website.Acclaim
      website.Mini
      WLANDemo
      ZTPDemo
    SSL
```

Figure 5. Directory Structure for a ZTP-Based Library System

Building and Running the SSL Demo

The following three procedures will help you to build the SSL Demo sample application, run it in a browser, and send an encrypted message between two eZ80 development platforms.

Build the SSL Demo Application

Observe the following procedure to build the SSL Demo application.

1. Launch the ZDS II Integrated Development Environment (IDE). From the Windows 7 **Start** menu, select the **All Programs** menu option. Scroll through the program and folder listings to the **Zilog ZDS II – eZ80Acclaim! A.B.C** folder item; click to expand this folder, then click the **ZDS II – eZ80Acclaim!_A.B.C** menu option.
2. From the **File** menu in ZDS II, select the **Open Project** menu option, and navigate to the location of the **SSLDemo** folder, as shown in Figure 4 (U.S.) or Figure 5 (international).
3. Within the **SSLDemo** folder, open the ZDS II project file that corresponds to the eZ80 development platform you are using.

   For example, if you are using the eZ80F910300KITG Development Kit, open the `ZTPSSLDemo_eZ80F910300KITG.zdsproj` project file.
4. Select the **Build** ⟶ **Rebuild All** menu option.
5. Download the executable file to the eZ80 Development Platform.

View an Embedded Website using SSL

Observe the following procedure to view an embedded website using SSL.

Note: If you have not previously used ZTP or have not run any of the sample projects, please refer to the *Hardware Setup* procedure in the Zilog TCP/IP Software Suite Quick Start Guide (QS0049) to become familiar with how to run ZTP sample programs.

1. During initialization, the `SSLDemo` project will display its IP address in the terminal emulation program (e.g., HyperTerminal). Record this IP address so that it can be used in the steps that follow.
2. On the PC, launch an SSL-enabled browser such as Internet Explorer. In the browser's URL bar, enter the following address:

   ```
   https:\\a.b.c.d
   ```

   In this URL, `a.b.c.d` is the IP address that was recorded in Step 1.

Note: To open a web page on a ZTP system without using SSL, simply enter `a.b.c.d` into the browser's URL bar to instruct the browser to retrieve the default web page using the HTTP protocol. A more formal method of specifying this URL is to use the syntax `http:\\a.b.c.d`, which explicitly tells the browser that the default web page is to be opened using the HTTP protocol. In the procedure above, the URL syntax was `https:\\a.b.c.d`, which directs the browser to open the default website using the HTTPS protocol which is simply HTTP running over SSL.

Send an Encrypted Message

Observe the following procedure to send an encrypted message between two eZ80 development platforms.

1. Build the SSL demo application for the eZ80 development platform that will operate as a server and download the executable.
2. Similarly, rebuild the application for the second eZ80 development platform that will operate in client mode. If necessary, change the MAC and IP addresses used by the client.
3. Start the server program by entering the following command on the server system's console:

   ```
   ssldemo server 5000
   ```

4. Send an encrypted (secret) message to the server by entering the following command on the client's console:

   ```
   ssldemo client 192.168.1.23:5000 “secret message”
   ```

Note: Ensure that you use the IP address of the target server in the command.

For additional information about building and running this sample program, refer to the ZTP Network Security SSL Plug-In Quick Start Guide (QS0059), which is available free for download on the Zilog website; it can also be found in the `docs` directory.

SSL Configuration

Before customizing the SSL demo project or adding SSL support to your existing ZTP application, see Table 2. Additional information is available in the ZTP Network Security SSL Plug-In Reference Manual (RM0047). The Default Configuration File column identifies the source file that contains the default setting for the indicated parameter.

The SSL demo project supplied with the ZTP Network Security SSL Plug-In already includes the default values for all of the configuration options. You can examine the SSL demo project to get a better understanding of how these configurable options can be customized.

Table 2. SSL Configuration Reference

Configurable Parameter	Options	Default Configuration File	Description
ZDSII project settings	ZDSII project configuration	*.zdsproj	Must specify Real-Time Kernel used, add SSL header files to Include Paths, and link SSL libraries.
SSL initialization	None	main.c	Mandatory
SSL handshake protocol initialization	• SSL2_ClientInit • SSL2_ServerInit • SSL3_ClientInit • SSL3_ServerInit • TLS1_ClientInit • TLS1_ServerInit	main.c	At least one handshake protocol must be initialized.
Digest algorithm selection	• MD5 • SHA1 • HMAC_MD5 • HMAC_SHA1	hash_conf.c	MD5 must be included for all versions of SSL. SSLv3 also requires SHA1. TLSv1 requires all digest algorithms.
Cipher algorithm selection	• RC4 • DES • 3DES • AES	cipher_conf.c	
PKI algorithm selection	• RSA • DSA • DH	pki_conf.c	PKI algorithm should match server's certificate.
Cipher suite configuration		ssl_conf.c	Valid combinations of digest, cipher, and PKI algorithms used to secure application level data.

Table 2. SSL Configuration Reference (Continued)

Configurable Parameter	Options	Default Configuration File	Description
EDH parameters		dh_param.c	Must be supplied to support Ephemeral Diffie-Hellman cipher suites in Server mode. Must also set an EDH function pointer to enable client or server EDH cipher suites.
Certificates	Not required for client only operation	Certificate.c	Server certificates must be accompanied by a private key.
Certificate verification		Certificate.c	User visible callback routine.
Signature verification	• TRUE • FALSE	ssl_conf.c	Determines whether SSL will verify digital signatures.
Session cache	• Size • Time-out	ssl_conf.c	Specifies whether client and server session information can be cached.
Diagnostic messages	• SSL_DEBUG_NONE • SSL_DEBUG_ERROR • SSL_DEBUG_WARNING • SSL_DEBUG_INFO	ssl_conf.c	Controls amount of diagnostic information displayed on the console.

SSL Configuration using ZDS II

This section explains how to configure the ZTP Network Security SSL Plug-In using ZDS II. The SSLDemo project supplied with the ZTP Network Security SSL Plug-In already includes all of these configuration steps.

ZDS II Project Settings

To use the ZTP Network Security SSL Plug-In in your ZDS II-based projects, the following items must be properly configured:

- Specification of the kernel
- Addition of SSL-related header files to the Include paths
- Specification of the SSL libraries to link with the project

Specify the Kernel

Ensure that the `SSL_OS_RZK` preprocessor symbol is included in the list of preprocessor symbols located in the **Preprocessor** tab of the **Project** → **Settings** → **C** menu.

Adding SSL Headers to the List of Include Paths

To access the SSL API, it is necessary to include the `SSL.h` header file in your application program. The `SSL.h` header file also includes other SSL-related header files. For your ZDS II project to build correctly, the path for all of the SSL-related header files must be specified in the Include paths input section in the **Preprocessor** tab of the **Project → Settings → C** menu. The default location of the SSL-related header files depends on the version of ZTP used; these paths are:

```
..\ZTP\Apps\Crypto\Inc for source package
..\ZTP\Inc for Library
```

Add the location of the above SSL-related header files to both the **User** and **Standard** text fields.

Linking the SSL Libraries

Before ZDS II can link a project using the SSL API, the SSL libraries to be used must be specified in the `Object/library modules` text field in either the **General** or **Input** categories on the **Linker** tab of the **Project Settings** menu. Zilog recommends that you always include all SSL-related libraries in the linker settings, and allow the ZDS II linker to decide which libraries are used and which are ignored. Adding unused libraries to the list of object/library modules does not increase the size of the final application. When specifying libraries, be sure to separate each library using a semicolon character (;).

The libraries listed in Table 3 are used to work with SSL.

Table 3. SSL Libraries

Library	Description
Crypto_SSL.lib	Contains SSL and Crypto files.
ZTPSSLCore.lib	Contains the socket files that are used for SSL.

SSL Initialization

Before attempting to initialize any of the SSL handshake protocols or using the SSL interface layer, the application program must call the `Initialize_SSL` API. This API activates the interface used by application programs to establish SSL sessions and securely transfer information. Because SSL uses the TCP transport layer, TCP must also be initialized to enable SSL functionality.

The `Initialize_SSL` API takes no parameters and will return either `SSL_SUCESS` or `SSL_FAILURE`. If `Initialize_SSL` is called more than once, all subsequent calls will fail.

The code fragment that follows shows an example of how to initialize the SSL interface:

```
{
  SSL_STATUS Status;
  Status = SSL_Initialize();
}
```

SSL Handshake Protocol Initialization

After calling the `Initialize_SSL` API, at least one of the SSL handshake protocols must be initialized. The ZTP Network Security SSL Plug-In is capable of supporting the following SSL protocols:

- SSL version 2 Client
- SSL version 2 Server
- SSL version 3 Client
- SSL version 3 Server
- TLS version 1 Client
- TLS version 1 Server

It is permissible to initialize one, two, or all six of these handshake protocols, but SSL will not function unless any one protocol is initialized.

To disable support for a particular handshake protocol, do not call its corresponding initialization function. Such a call will prevent the ZDS II linker from including code for that particular handshake protocol in the final application image.

Table 4 shows the relationship between the SSL handshake protocols and their initialization functions.

Table 4. SSL Handshake Protocol Functions

SSL Handshake Protocol	Initialization Function
SSL version 2 Client	SSL2_ClientInit
SSL version 2 Server	SSL2_ServerInit
SSL version 3 Client	SSL3_ClientInit
SSL version 3 Server	SSL3_ServerInit
TLS version 1 Client	TLS1_ClientInit
TLS version 1 Server	TLS1_ServerInit

Each of the `xxxs_ClientInit` APIs is a null function returning a variable of type `SSL_STATUS`. The `TLS1_ClientInit` API is shown in the following code fragment; the `SSL2_ClientInit` and `SSL3_ClientInit` APIs have the same format.

```
SSL_STATUS TLS1_ClientInit( void );
```

Each server initialization function takes two parameters and returns a status code. To illustrate these parameters, the function prototype for the `TLS1_ServerInit` API is shown in the following code fragment. The same syntax also applies to the `SSL2_ServerInit` and `SSL3_ServerInit` functions.

```
SSL_STATUS
TLS1_ServerInit
(
CERT_CHAIN           * pCertChain,
ASN1_ENC_DATA * pDheParams
);
```

The first of these parameters is a reference to the server's certificate chain, which is a list of X.509 certificates beginning with the server's certificate and followed by the certificate of each intermediate certificate authority that signed the previous certificate. The certificate chain ends with a self-signed root certificate issued by the certificate authority.

The second parameter is a pointer to the Diffie-Hellman parameters (the prime modulus, *p* and the generator, *g*) that the server will use for *Ephemeral Diffie-Hellman (EDH) cipher suites*. SSL clients will receive their Ephemeral Diffie-Hellman parameters from the server to which they are attempting to establish a connection. If support for Ephemeral Diffie-Hellman parameters is not required for either the TLS1 or SSL3 server, then this parameter is set to `NULLPTR` on the corresponding `xxxx_ServerInit` function call.

Note: SSL version 2 does not support Ephemeral Diffie-Hellman cipher suites; therefore this parameter should always be `NULLPTR` when calling the `SSL2_ServerInit` API.

It is permissible to use different certificate chains and Ephemeral Diffie-Hellman parameters on each of the handshake protocol initialization calls. In some cases, this usage is mandatory. For example, if the TLSv1 server has been issued a DSA certificate, this server will only be able to SSL sessions using EDH cipher suites. But if an SSLv2 server is also initialized, then that server must have an RSA certificate. Therefore, in this example, the SSLv2 and TLSv1 servers must use different certificate chains.

The SSL demo project included with the ZTP Network Security SSL Plug-In contains a file `dh_params.c` containing the Ephemeral Diffie-Hellman parameters used on the calls to `TLS1_ServerInit` and `SSL3_ServerInit`. The certificate chains shared by these server is contained in a file named `Certificate.c`.

Client Mode or Server Mode Support

When each of the SSL handshake protocols is initialized (see the SSL Handshake Protocol Initialization section on page 22), the first parameter on the initialization call specifies whether the protocol supports the client and/or server mode of operation.

Notes: 1. SSL sessions are always initialized by clients. Servers will only wait passively for connection attempts from remote clients.

2. The SSL handshake protocols implemented in the ZTP Network Security SSL Plug-In are capable of operating simultaneously. For example, it is possible that an SSLv3 client session is being established at the same time as a TLSv1 server session and a TLSv1 client session. However, when multiple sessions are established at the same time, it takes longer for all sessions to be established than if they had been established serially.

Digest Algorithm Selection

The SSL handshake protocols use digest algorithms for many purposes, including generating and verifying message authentication codes, generating session keys, and verifying digital signatures. The ZTP Network Security SSL Plug-In recognizes the following four digest (or hash) algorithms:

- MD5
- SHA1
- HMAC_MD5
- HMAC_SHA1

Depending on the SSL handshake protocols used and the configuration of cipher suites, some of these digest algorithms can be removed from the project to reduce code size. This removal can be performed by modifying `HashGen` array in the `hash_conf.c` file.

The default setting of the `HashGen` array is shown in the following code fragment.

```
HASH_NEW              HashGen[ SSL_MAX_HASH ] =
{
  NullHash_New,
  MD5_New,
  HMAC_MD5_New,
  SHA1_New,
  HMAC_SHA1_New
};
```

Note: Each of the four supported digest algorithms has an entry that follows the `NullHash_New` function pointer.

Each entry in the array is a function pointer that is used to initialize a data structure that the SSL handshake protocols use to perform digest operations. `NullHash` does not perform any useful function, but it must be included in the `HashGen` array for proper operation of the SSL protocol.

The ordering of entries in the `HashGen` array is not arbitrary, and is determined by the ordinal values of the following macros (see the `ez80_hash.h` header file); the values of these macros must not be altered.

```
#define SSL_HASH_NULL            0
#define SSL_HASH_MD5             1
#define SSL_HASH_HMAC_MD5        2
#define SSL_HASH_SHA1            3
#define SSL_HASH_HMAC_SHA1       4
```

Table 5 shows which digest algorithms are required for each of the SSL handshake protocols. If the application uses combinations of protocols, select the last row in the table that matches one of the SSL handshake protocols used. For example, if your project requires the SSLv2 and TLSv1 handshake protocols, then the digest algorithms corresponding to the TLSv1 handshake protocol must appear in the `HashGen` array.

Table 5. Mandatory Digest Algorithm by SSL Protocol Version

SSL Handshake Protocol version	Mandatory Digest Algorithms
SSLv2	MD5
SSLv3	MD5, SHA1
TLSv1	MD5, SHA1, HMAC_MD5, HMAC_SHA1

If a digest algorithm is not required, replace the corresponding entry in the `HashGen` array with `NullHash_New`. For example, in an application required to support SSLv2 and SSLv3, it is not necessary to include the `HMAC_MD5` or `HMAC_SHA1` digest algorithms; therefore the project can be made slightly smaller by using the following `HashGen` array.

```
HASH_NEW                HashGen[ SSL_MAX_HASH ] =
{
  NullHash_New,
  MD5_New,
  NullHash_New,
  SHA1_New,
  NullHash_New
};
```

Care must be taken while removing the SHA1 digest algorithm. Sometimes this algorithm is used in the process of signing digital certificates. Therefore, if the SHA1 digest algorithm is not configured into the system and the SSL protocol must either generate or verify a signature using the SHA1 algorithm, it will not be possible to complete the operation. As a result, an SSL session will be prevented from becoming established.

It is important to keep the `HashGen` array synchronized with the table of cipher suites referenced by the `pSSL2_CipherSuites`, `pSSL3_CipherSuites` and `pTLS1_CipherSuites` pointers. For example, if the `SHA1_New` function pointer is replaced with `NullHash_New`, then the SHA1 digest algorithm will not be included in the application. Therefore, if any of the cipher suite tables contains an entry which uses SHA1, such as `TLS_RSA_WITH_3DES_EDE_CBC_SHA`, these cipher suites must be disabled as they will not function properly without the SHA1 algorithm. For more information about this topic, see the Cipher Suite Configuration section on page 31.

Cipher Algorithm Selection

The SSL handshake protocols use symmetric cipher algorithms to encrypt and decrypt application level data transferred through SSL. With symmetric ciphers, both the client and server use the same set of keys to encrypt and decrypt data. These symmetric keys are changed each time the client and server establish a new session. These keys are derived from information exchanged during the execution of the (asymmetric) PKI algorithm during the establishment of a session.

The ZTP Network Security SSL Plug-In recognizes the following four cipher algorithms:

- RC4 (128-bit key)
- DES (56-bit key)
- 3DES (168-bit key)
- AES (128-bit key or 256-bit key)

Note: Each of these four cipher algorithms has an entry that follows the `NullCipher_New` function pointer.

An SSL session is established using any one of these cipher algorithms. However, not all SSL clients and servers implement the same set of ciphers. By supporting multiple cipher algorithms, there is a possibility that the client and server will be able to determine at least one common algorithm that can be used to encrypt data. However, if a weak cipher algorithm is included in the set of supported cipher algorithms, then it is possible that at some point, a session could be established with the weaker algorithm.

The ZTP Network Security SSL Plug-In uses a global array, named the `CipherGen` array, that determines which symmetric ciphers are available for encrypting data. This array is located in the `cipher_conf.c` configuration file. The default setting of the `CipherGen` array is shown in the following code fragment.

```
CIPHER_NEW          CipherGen[ SSL_MAX_CIPHERS ] =
{
  NullCipher_New,
  RC4_New,
  DES_New,
  DES3_New,
  AES_New
};
```

Each entry in the array is a function pointer used to initialize a data structure that the SSL handshake protocols use to perform encryption and decryption operations. `NullCipher` does not perform any useful function, but it must be included in the `CipherGen` array for proper operation of the SSL protocol.

The ordering of entries in the `CipherGen` array is not arbitrary, and is determined by the ordinal values of the following macros (see the `ez80_cipher.h` header file); the values of these macros must not be altered.

```
#define SSL_CIPHER_RC4              1
#define SSL_CIPHER_DES              2
#define SSL_CIPHER_3DES             3
#define SSL_CIPHER_AES              4
```

If an application does not require the use of a particular cipher, its entry in the `CipherGen` array can be replaced with the `NullCipher_Init` function pointer. This pointer has the effect of causing the linker to remove the cipher algorithm from the generated program image. For example, if an application does not use the AES cipher, the `CipherGen` array can be reconfigured, as shown in the following code fragment:

```
CIPHER_NEW          CipherGen[ SSL_MAX_CIPHERS ] =
{
  NullCipher_New,
  RC4_New,
  DES_New,
  DES3_New,
  NullCipher_New
};
```

It is important to keep the `CipherGen` array synchronized with the table of cipher suites referenced by the `pSSL2_CipherSuites`, `pSSL3_CipherSuites`, and `pTLS1_CipherSuites` pointers. For example, if the `DES3_New` function pointer is replaced with `NullCipher_New`, then the 3DES cipher algorithm will not be included in the application. Therefore, if any of the cipher suite tables contains an entry which uses 3DES – such as `TLS_RSA_WITH_3DES_EDE_CBC_SHA` – these cipher suites must be disabled because they will not function properly without the 3DES algorithm. For more information about this topic, see the Cipher Suite Configuration section on page 31.

Note: The AES cipher was standardized after the SSL specifications were created. Therefore, there are no defined cipher suites in the SSLv2, SSLv3 or TLSv1 specifications that use AES. However, changes to the TLSv1 specification have included cipher suites that use either the 128-bit or 256-bit AES algorithm. The ZTP Network Security SSL Plug-In defines several cipher suites for TLSv1 and SSLv3 using AES, but does not include any definitions for SSLv2 cipher suites using AES. Because AES is relatively new to SSL, it may be difficult to find third party applications supporting AES-based cipher suites. At the date of publication of this document, for example, Microsoft Internet Explorer v6.0 did not include AES support.

PKI Algorithm Selection

A public key infrastructure (PKI) allows an insecure network to exchange data securely via authentication and privately via encryption. PKI systems employ public key cryptography in which a public and private key pair is used to perform cryptographic operations.

In public key cryptography, one entity holds the private key in secrecy; while the public key is freely distributed. Public key algorithms allow entities to securely arrive at a common shared secret without having any prior knowledge of each other.

In contrast, private key systems require all parties to be in possession of a common shared secret before secure communication begins. As the number of participants in private key systems increases, it becomes harder to keep the shared secret private. Once a secret is compromised, secure communication is no longer assured. In public key systems, only one device must keep the private key a secret regardless of the number of participants. The downside of public key cryptography is that its asymmetric algorithms tend to be more computationally intensive than private key symmetric algorithms.

SSL servers are required to be in possession of a public and private key pair. The server's public key is placed into a certificate that is signed and validated by a trusted third party. During the establishment of an SSL session, the client receives a copy of the server's certificate and, therefore, the public key. A field within the certificate indicates which public key algorithm can be used to arrive at a shared secret that will be used to derive the shared symmetric key(s) used to encrypt data exchanged between the parties.

The ZTP Network Security SSL Plug-In supports the following three public key algorithms:

- RSA encryption
- DSA signature
- Diffie-Hellman key agreement

The most popular algorithm used with SSL is RSA encryption, which is the only key exchange algorithm supported by SSLv2. On their own, DSA signatures cannot be used to establish a shared secret, but the DSA algorithm is used to sign Ephemeral Diffie-Hellman parameters, thereby allowing the Diffie-Hellman key agreement algorithm to arrive at a shared secret.

A global array, `PkiGen`, determines which public key algorithms are available for use by the SSL handshake protocols. This array is located in the `pki_conf.c` configuration file. The default setting of the `PkiGen` array is shown in the following code fragment.

```
PKI_Init            PkiGen[SSL_MAX_PKI] =
{
  NullPki_init,
  rsa_init,
  dsa_init,
  dh_init
};
```

In the above code, note that a call to each of the three supported PKI algorithms appears below the `NullPki_init` function pointer. Indeed, each entry in the array is a function pointer that is used to initialize a data structure that the SSL handshake protocols use during key exchange processing. The `NullPki` algorithm does not perform any useful function, but it must be included in the `PkiGen` array for proper operation of the SSL protocol.

The ordering of entries in the `PkiGen` array is not arbitrary, and is determined by the ordinal values of the following macros (see the `ez80_pki.h` header file); the values of these macros must not be altered.

```
#define SSL_PKI_ID_RSA              1
#define SSL_PKI_ID_DSA              2
#define SSL_PKI_ID_DH               3
```

If an application does not use a particular PKI algorithm, its entry in the `PkiGen` array can be replaced with the `NullPki_init` function pointer. This pointer has the effect of causing the linker to remove the PKI algorithm from the generated program image. For example, if an application does not use the Diffie-Hellman key agreement algorithm, the `PkiGen` array is reconfigured, as shown in the following code fragment:

```
PKI_Init                PkiGen[SSL_MAX_PKI] =
{
  NullPki_init,
  rsa_init,
  dsa_init,
  NullPki_init
};
```

It is important to keep the `PkiGen` array synchronized with the tables of cipher suites referenced by the `pSSL2_CipherSuites`, `pSSL3_CipherSuites`, and `pTLS1_CipherSuites` pointers. For example, if the `rsa_init` function pointer is replaced with `NullPki_init`, then the RSA algorithm will not be included in the application. Therefore, if any of the cipher suite tables contains an entry which uses RSA, such as `TLS_RSA_WITH_3DES_EDE_CBC_SHA`, these cipher suite must be disabled because they do not function properly without the RSA algorithm. For more information about this topic, see the Cipher Suite Configuration section that follows.

Notes:
1. If the ZTP Network Security SSL Plug-In has been configured to verify signatures, it could become necessary to include both RSA and DSA in the `PkiGen` array, because these algorithms are required to verify signatures generated with the same algorithm.
2. The use of Diffie-Hellman certificates is extremely rare. Therefore, the `dh_init` function pointer is usually replaced with `NullPki_init`. This replacement will not prevent the SSLv3 and TLSv1 protocols from using the Diffie-Hellman key agreement algorithm with ephemeral parameters. Use of the Diffie-Hellman algorithm with ephemeral parameters is controlled by the `pDheInit` function pointer in the `pki_conf.c` configuration file. Therefore, to completely remove Diffie-Hellman from the application, the `dh_init` entry in the `PkiGen` array must be replaced with `NullPki_init`, and the `pDheInit` function pointer must be set to `NULLPTR`. For more information about this topic, see the EDH Parameters section on page 37 .

Practical Considerations for SSL Servers and Clients

When the ZTP Network Security SSL Plug-In is operating in server mode, the server must have a certificate indicating which public key algorithm is required. The corresponding algorithm initialization function pointer must be in the `PkiGen` array.

When the ZTP Network Security SSL Plug-In is operating in client mode, it does not know the type of certificate a server possesses. If the server presents a certificate using a public key algorithm which is not included in the `PkiGen` table, then the client will be unable to establish a session with the server. The greatest flexibility is afforded by including all possible algorithms in the `PkiGen` array. However, such an inclusion comes at the expense of increasing the code size of the application due to public key algorithms that are rarely used. Due to the overwhelming popularity of the RSA algorithm, Zilog recommends that the RSA algorithm always be included in the `PkiGen` array.

Cipher Suite Configuration

During the establishment of an SSL session, the client and server determine a *3-tuple* of PKI algorithm, symmetric cipher algorithm and digest algorithm that is used to secure communications. This 3-tuple is called a *cipher suite*. Because SSL supports different PKI, cipher and digest algorithms, there are many possible combinations of cipher suites.

The `ssl_conf.c` configuration file contains tables of `SSL_CS_INFO` structures that define a set of cipher suites that can be supported by the ZTP Network Security SSL Plug-In. Individual entries in these tables can be removed or disabled to prevent the cipher suite from being selected during the establishment of a session. These tables also determine the minimum set of PKI, cipher and digest algorithms that must be included in the `PkiGen`, `CipherGen` and `HashGen` arrays.

To understand these relationships, first understand the structure of a single cipher suite. The `SSL_CS_INFO` data structure is shown in the following code fragment.

```
typedef struct SSL_CS_INFO
{
  SSL_WORD          CipherSuite;
  SSL_BYTE          KeyAlg;
  SSL_BYTE          CipherAlg;
  SSL_BYTE          HashAlg;
  SSL_BOOL          IsExport;
  SSL_BYTE          KeySize;
  SSL_BYTE          IVSize;
  SSL_BYTE          MacSize;
  SSL_BOOL          IsValid;
} SSL_CS_INFO;
```

The `CipherSuite` entry is a two-byte code that the SSL specification defines to identify the standard cipher suites. Each code also features a long mnemonic. The cipher suites that the ZTP Network Security SSL Plug-In is capable of supporting can be found in the `CipherSuite.h` header file; some examples are shown in the following code fragment.

```
#define TLS_RSA_WITH_RC4_128_MD5                0x0400
```

```
#define TLS_RSA_WITH_AES_128_CBC_SHA           0x2F00
#define TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA       0x0D00
#define TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA      0x1300
```

The `KeyAlg` entry identifies the algorithm that the ZTP Network Security SSL Plug-In uses to arrive at a shared secret between the client and server when using this cipher suite. The macros that the ZTP Network Security SSL Plug-In uses for the `KeyAlg` codes are defined in the `ez80_pki.h` header file and are shown in the following code fragment.

```
#define SSL_PKI_RSA                 1
#define SSL_PKI_DH                  2
#define SSL_PKI_DHE_RSA             3
#define SSL_PKI_DHE_DSS             4
```

The `CipherAlg` entry identifies one of the supported cipher algorithms (see the `ez80_cipher.h` header file).

```
#define SSL_CIPHER_NULL             0
#define SSL_CIPHER_RC4              1
#define SSL_CIPHER_DES              2
#define SSL_CIPHER_3DES             3
#define SSL_CIPHER_AES              4
```

The `HashAlg` entry identifies one of the supported digest algorithms (see the `ez80_hash.h` header file).

```
#define SSL_HASH_NULL               0
#define SSL_HASH_MD5                1
#define SSL_HASH_HMAC_MD5           2
#define SSL_HASH_SHA1               3
#define SSL_HASH_HMAC_SHA1          4
```

The `IsExport` entry indicates whether this cipher suite can be exported for use outside the United States. Only those cipher suites that contain the word EXPORT in the mnemonic can be used in products outside the United States without government approval. All exportable SSL cipher suites implemented by the ZTP Network Security SSL Plug-In use an effective 40-bit symmetric key and restrict the modulus in the key exchange algorithm to a maximum of 512 bits[2].

The `KeySize` entry specifies the number of bytes in the symmetric key that are used by the cipher algorithm.

2. Although these limits are below current United States export law requirements, source code customers are advised to seek government counsel before modifying the SSL protocol to allow longer keys in exported cipher suites.

Note: For export cipher suites using a 40-bit effective symmetric key, the cipher algorithm still uses the full key length. The distinction is that only 40 bits of the symmetric key are protected during the key exchange algorithm for export ciphers.

`IVSize` denotes the length, in bytes, of the initialization vector used by the cipher algorithm. Block ciphers operating in cipher-block chaining mode (CBC) require an initialization vector.

The `MacSize` denotes the length, in bytes of the message authentication code that the SSL record layer generates on each outbound record and verifies on each inbound record. The MAC is constructed using the digest algorithm specified by the `HashAlg` identifier.

The `IsValid` flag determines whether or not the cipher suite definition is used while determining a compatible set of cipher suites between the SSL client and server. If the `IsValid` flag is set to FALSE, then the cipher suite is not used during negotiations. This setting allows applications to dynamically enable or disable cipher suites, as required.

Given a cipher suite mnemonic, it is simple to translate it into binary information which can be used by the SSL handshake protocols. For example, the `TLS_RSA_WITH_RC4_128_MD5` mnemonic can be characterized as:

- The cipher suite is designed for use with the TLS handshake protocol.
- The RSA algorithm is used to exchange a secret key.
- RC4 is used to encrypt/decrypt all communications once the session is established. The RC4 algorithm implemented in SSL uses a 128-bit symmetric key.
- MD5 is used as the message digest algorithm to generate MAC codes.

The corresponding `SSL_CS_INFO` structure that contains the same information is shown in the following code fragment:

```
{
  TLS_RSA_WITH_RC4_128_MD5,
  SSL_PKI_RSA,
  SSL_CIPHER_RC4,
  SSL_HASH_MD5,
  FALSE,
  RC4_128_KEY_SIZE_BYTES,
  RC4_IV_SIZE_BYTES,
  MD5_HASH_SIZE_BYTES,
  TRUE
};
```

Because the mnemonic does not contain the word EXPORT, the `IsExport` flag is set to FALSE. Additionally manifest constants are used for the sizes of the `KeySize`, `IVSize`, and `MacSize` fields. Lastly, the `IsValid` flag is set to TRUE so that this cipher suite can be used for establishing a session.

Cipher Suite Tables

The first step in establishing an SSL session is for the client and server to determine a common cipher suite. Both the client and server must be capable of supporting the exact same cipher suite, or else the session will not be established. If the client supports only `TLS_RSA_WITH_RC4_128_MD5` and the server supports only `TLS_RSA_WITH_RC4_128_SHA`, the client and server cannot establish an SSL session.

To facilitate the establishment of a session, it is advantageous if both parties are capable of supporting multiple cipher suites. This type of support increases the chance that at least one match will be found. With the ZTP Network Security SSL Plug-In, this task is accomplished by creating tables of `SSL_CS_INFO` structures that define a set of cipher suites supported by the SSL handshake protocols. Each implemented SSL handshake protocol has a global variable, defined in `ssl_conf.c`, that references its table of cipher suites. These variables are named:

- `pSSL2_CipherSuites`
- `pSSL3_CipherSuites`
- `pTLS1_CipherSuites`

A second set of global variables specifies how many entries are present in each of the cipher suite tables. These variables are named:

- `NumSSL2_CipherSuites`
- `NumSSL3_CipherSuites`
- `NumTLS1_CipherSuites`

The following code fragment shows a sample cipher suite table for the TLSv1 handshake protocol. For clarity, only the mnemonic of the `SSL_CS_INFO` structure is shown, followed by an ellipsis.

```
SSL_CS_INFO         TLS1_CipherSuites[] =
{
  { TLS_NULL_WITH_NULL_NULL, ... },
  { TLS_RSA_WITH_RC4_128_MD5, ... },
  { TLS_RSA_WITH_DES_CBC_SHA, ... }
};
SSL_BYTE            NumTLS1_CipherSuites =
sizeof(TLS1_CipherSuites) / sizeof(SSL_CS_INFO);
SSL_CS_INFO         *pTLS1_CipherSuites   = TLS1_CipherSuites;
```

Note: The first entry in every cipher suite table must indicate a NULL cipher suite; i.e., one that uses the NULL PKI algorithm, the NULL cipher algorithm and the NULL digest algorithm. This cipher suite must never be enabled (i.e., `IsValid` is set to FALSE). It is included in the cipher suite because it describes the session's initial state (operation on a completely unsecured channel).

The ordering of cipher suites within each table is significant. Entries appearing higher in the table are preferred over entries appearing lower in the table. For example, in the sample cipher suite table above, it is possible that both the client and server support both cipher suites; however, because the `TLS_RSA_WITH_RC4_128_MD5` entry appears before `TLS_RSA_WITH_DES_CBC_SHA`, preference will be given to the `TLS_RSA_WITH_RC4_128_MD5` cipher suite. When the corresponding ZTP Network Security SSL Plug-In SSL handshake protocol is operating as a server, it selects the first matching entry in the cipher suite table that matches the list of cipher suites supplied by the client. When the corresponding SSL protocol is operating in client mode, it orders its list of supported cipher suites in the same order as they appear in the cipher suite table, thus indicating the order of preference to the server. In either situation, all cipher suites in the table for which the `IsValid` flag is FALSE are ignored.

Synchronizing PKI, Cipher and Digest Configurations

After the cipher suite tables are created, it is easy to determine the minimal set of PKI algorithms, cipher algorithms and digest algorithms that must be configured in the `PkiGen`, `CipherGen`, and `HashGen` arrays. For example, to determine what entries must exist in the `CipherGen` array to support all cipher suites for which the `IsValid` flag is set to TRUE, note each unique entry in the `CipherAlg` field. Suppose the cipher suites all used `SSL_CIPHER_NULL`, `SSL_CIPHER_RC4` or `SSL_CIPHER_DES`. As a result, the `CipherGen` array could be modified, as shown in the following code fragment, because the 3DES and AES cipher algorithms will not be required.

```
CIPHER_NEW           CipherGen[ SSL_MAX_CIPHERS ] =
{
  NullCipher_New,
  RC4_New,
  DES_New,
  NullCipher_New,   // 3DES not required
  NullCipher_New    // AES not required
};
```

Configuring the `HashGen` array is slightly more complicated, because the `HMAC_MD5` and `HMAC_SHA1` hashes are always used by TLSv1 – even though they never appear in the

CipherGen table. For additional information about configuring the HashGen array, see Table 5 on page 25.

Configuring the PkiGen array is difficult. Use the values listed in Table 6 to determine the minimum set of PKI algorithms required based on the KeyAlg field in all cipher suite entries.

Table 6. PKI Algorithm Requirements by Cipher Suite

KeyAlg Value From Cipher Suite	Required PKIGen Entry	Required pDheInit Setting
SSL_PKI_RSA	rsa_init	NULLPTR
SSL_PKI_DH	dh_init	NULLPTR
SSL_PKI_DHE_RSA	rsa_init	dhe_init
SSL_PKI_DHE_DSS	dsa_init	dhe_init

Notes: 1. When RSA export cipher suites are used, the ZTP Network Security SSL Plug-In will abort the establishment of a session if the RSA modulus exceeds the export limit regarding public key size.

2. Any cipher suite containing the text *DHE* uses Ephemeral Diffie-Hellman (EDH) parameters to arrive at a shared secret between the client and the server. Therefore, the pDheInit function pointer must reference the dhe_init routine, or else Ephemeral Diffie-Hellman cipher suites cannot be supported. The difference between a Diffie-Hellman (DH) certificate and DHE parameters is that the private and public Diffie-Hellman values never change when a DH certificate is used. In contrast, when DHE parameters are used, the private and public values are changed each time a new session is established.

3. When DSS certificates (using the DSA signature algorithm) are employed, EDH key exchange is always performed. This situation exists because the DSA algorithm cannot be used to establish a shared secret; it can only be used to digitally sign some other datum. Therefore, DHE_DSS cipher suites use Ephemeral Diffie-Hellman parameters to arrive at a shared secret, and these parameters are signed using the public key contained in the DSS certificate.

RSA certificates are used for encryption and signatures. Cipher suites using RSA for key exchange through RSA encryption contain text such as _RSA_WITH_ or _RSA_EXPORT_WITH_. Cipher suites using Ephemeral Diffie-Hellman parameters signed with RSA use text such as _DHE_RSA_.

In general, when a cipher suite contains two public key algorithms (for example, TLS_DHE_RSA_WITH_DES_CBC_SHA), the first public key algorithm identifies the key exchange algorithm (DHE in this example). The second public key algorithm identifies the algorithm used for authentication (RSA in this example). Because RSA can be used for both key exchange and authentication, the text *RSA* only appears once in the cipher suite mnemonic. This feature represents another advantage of using RSA certificates; i.e., only one (computationally-intensive) public key operation must be performed to authenticate the server and arrive at a shared secret, as opposed to cipher suites requiring the use of two public key algorithms.

As indicated above, SSL primarily uses two public key algorithms to generate and verify DSA and RSA signatures. Because the use of RSA certificates is more prevalent than DSS (i.e., DSA) certificates, it is not required that the `dsa_init` function pointer be included in the `PkiGen` array. However, in cases where a server's certificate is signed using an issuer's DSA private key, it will not be possible to verify the signature unless the `dsa_init` function pointer is included in the `PkiGen` array. Furthermore, if signature verification is enabled (see the Signature Verification section on page 50), it will not be possible to establish a session.

EDH Parameters

When an Ephemeral Diffie-Hellman cipher suite is used to secure a communication channel between the SSL client and server, the server must supply DH parameters to the client in a handshake message. Ephemeral Diffie-Hellman cipher suites contain the text `_DHE_` in the cipher suite mnemonic; e.g., `TLS_DHE_RSA_WITH_DES_CBC_SHA`. The DH parameters contain two values necessary to complete the Diffie-Hellman key agreement algorithm: the prime modulus, *p*, and the generator, *g*. The message that is sent also contains the server's public value.

In the ZTP Network Security SSL Plug-In implementation, the SSL server selects a new private key and generates a public value prior to sending the Ephemeral Diffie-Hellman parameters and the public value to the client. However, the server always uses the same DH parameters (*p* and *g*). A reference to these DH parameters must be supplied to the server on the `TLS1_ServerInit` or `SSL3_ServerInit` APIs, as shown in the following TLSv1 initialization call example:

```
TLS1_ServerInit( &CertChain, &DheParams );
```

By default, the SSLv3 and TLSv1 servers in the ZTP Network Security SSL Plug-In use the same set of DH parameters, but this usage is not mandatory. Because SSLv2 cipher suites do not use Ephemeral Diffie-Hellman cipher suites, the final parameter on the `SSL2_Init` API is always specified as `NULLPTR`.

In the default configuration of the ZTP Network Security SSL Plug-In, the Ephemeral Diffie-Hellman parameters are contained in the `dh_params.c` configuration file, as shown in the following code fragment:

```
SSL_BYTE DH_Params_Pem[] = {"\
MIGKAkEA3uxiDPwIuoU6r22inWehs84FBTvrD8bQufdCltw6RAoV+DM5PHkyMLoH\
KEThy65yDANqA0s4tukYX+jEg98IFQJBAKK+9mbWv9G6WqQExbjrjxKUJG863bYR\
QlwmO9kd6hs6rQDa1g1E5UQ9SOrUcs6cLGzuSQYE+0K8G7UEknvAKTYCAgCg"};

ASN1_ENC_DATA       DheParams =
{
  PEM_ENCODED_DATA,
  sizeof(DH_Params_Pem)-1,
  DH_Params_Pem
};
```

Note: The *p* and *g* values must be encoded as a sequence of two ASN.1 DER integers. The first integer is the value of the prime modulus, *p*, and the second integer is the value of the generator, *g*. The SSL library assumes that *p* and *g* have been chosen appropriately. The ASN.1 DER data can optionally be Base64-encoded (as shown in the example above).

The variable used to specify the DH parameters must be of type `ASN1_ENC_DATA` block. The encoding member of the data structure indicates whether the DH parameters are `DER_ENCODED_DATA` (i.e., DER-encoded ASN.1 data) or `BASE64_DER_ENCODED_DATA` (i.e., Base64 DER-encoded ASN.1 data). Base64 DER-encoded ASN.1 data is also known as PEM encoding and, therefore, the encoding member of the `ASN1_ENC_DATA` variable can be specified as `PEM_ENCODED_DATA`.

The `length` and `pData` members of the `ASN1_ENC_DATA` structure indicate the number of bytes in the encoded data and the address of the first byte of the encoded data.

How to Generate Ephemeral Diffie-Hellman Parameters

The ZTP Network Security SSL Plug-In does not include any utilities to generate Diffie-Hellman parameters. Third-party utilities must be used to generate Ephemeral Diffie-Hellman parameters, if required; Zilog does not recommend or endorse any such utilities. This section describes how to generate Diffie-Hellman parameters using the `dhparam` command in OpenSSL.

To generate a Diffie-Hellman parameter, enter the following command at the `OpenSSL` command prompt:

```
OpenSSL> dhparam -text -out dh_param.txt 512
```

This command will produce an output file named `dh_param.txt` that contains the prime modulus and generator in text format, as well as a Base64-encoded ASN.1 DER data block containing the DH parameters. The maximum length of the DH modulus is 512 bits. An example of the contents of the text file is shown in the following code fragment:

```
Diffie-Hellman-Parameters: (512 bit)
```

```
  prime:
  00:84:5f:92:80:12:59:11:5a:5d:22:84:e9:8d:6e:
  fc:1b:6b:e4:7d:bb:76:97:57:07:c1:9a:4d:1f:ea:
  88:ae:d5:13:08:5a:00:9a:78:a2:28:47:aa:f6:90:
  ce:5d:cd:75:01:cc:9c:89:7a:79:4d:af:37:c1:ad:
  ba:74:3d:12:3b
  generator: 2 (0x2)
-----BEGIN DH PARAMETERS-----
MEYCQQCEX5KAElkRWl0ihOmNbvwba+R9u3aXVwfBmk0f6oiu1RMIWgCaeKIoR6r2
kM5dzXUBzJyJenlNrzfBrbp0PRI7AgEC
-----END DH PARAMETERS-----
```

To use this output with the ZTP Network Security SSL Plug-In, cut and paste the two lines of text between the BEGIN DH PARAMETERS and END DH PARAMETERS delimiters mentioned above, and place it into an array that is referenced by the `pData` member of the `ASN1_ENC_DATA` structure. The following code fragment shows how to instantiate a variable of type `ASN1_ENC_DATA` that uses these DH parameters.

```
SSL_BYTE DH_Params_Pem[] = {"\
MEYCQQCEX5KAElkRWl0ihOmNbvwba+R9u3aXVwfBmk0f6oiu1RMIWgCaeKIoR6r2\
kM5dzXUBzJyJenlNrzfBrbp0PRI7AgEC"};
ASN1_ENC_DATA       DheParams =
{
  PEM_ENCODED_DATA,
  sizeof(DH_Params_Pem)-1,
  DH_Params_Pem
};
```

By default, the OpenSSL `dhparam` command will generate DH parameters with a generator of two or five. If larger generators are required, use the `-dsaparam` option, as shown in the following code fragment:

```
OpenSSL> dhparam -text -dsaparam -out dh_param.txt 512
```

Modulus Length

As the number of bits in the DH modulus is increased, it becomes more difficult for attackers to guess the DH shared secret generated by the algorithm. However, this increase also increases the amount of time it takes for the ZTP Network Security SSL Plug-In to compute results using the DH key exchange algorithm.

Additionally, if export cipher suites are enabled, it is important to ensure that the modulus does not exceed export requirements. The original SSLv3 and TLSv1 cipher suites required the DH modulus of export cipher suites to be less than or equal to 512 bits to conform with United States export regulations at that time. Since then, export regulations have been relaxed to allow the export of 1024-bit public keys (and 56-bit symmetric keys);

however, the ZTP Network Security SSL Plug-In does not currently recognize extended cipher suites that employ 1024-bit public keys.

Certificates

Before describing where certificates are configured in the ZTP Network Security SSL Plug-In, you can review the role of certificates in SSL.

Background

During the establishment of a session, X.509 certificates are mainly used to authenticate the server, and are optionally used to authenticate the client. The SSLv3 and TLSv1 specifications also define a set of anonymous cipher suites in which neither party is authenticated. The ZTP Network Security SSL Plug-In does not support client authentication, nor does it support anonymous cipher suites. Therefore, only SSL servers are required to possess an X.509 certificate.

X.509 certificates contain information that identifies the entity for which the certificate was issued (referred to as the *subject* of the certificate) and information about the entity that issued the certificate (the issuer). Certificates are valid only for a certain period of time. Each certificate contains two time stamps. The first time stamp specifies the start of a certificate's validity period. The second time stamp identifies the time at which the certificate expires. Certificates used outside this time period are to be treated as invalid. For the purpose of establishing an SSL session, the two most important items in the certificate are the server's public key and the certificate signature.

When a certificate is created, the issuer asserts that the subject of the certificate is in possession of a private key corresponding to the public key in the certificate. The issuer also asserts that it has performed some level of verification (indeed, perhaps none) of the subject's identity. The exact information required by an issuer to verify a subject's identity will vary between issuers. This scenario is analogous to various certificates used by people, such as a library card or a passport. The background checks performed by the respective issuers are not necessarily identical. Therefore, when presenting either of these certificates to prove one's identity, one of these certificates may be more accepted (i.e., trusted) because the issuer has more credibility.

Trust relationships form the basis of SSL authentication. When an SSL server presents a client with its digital certificate, the client performs basic integrity checks on the certificate (for example, it may check whether the entity presenting the certificate is the same as the subject, or if the certificate has expired). However, a forged certificate could easily pass these basic integrity checks. Therefore, every X.509 certificate is signed by the issuer (using the issuer's private key). A client that is in possession of the issuer's certificate can use the issuer's public key to verify the authenticity of the certificate presented by the subject (i.e., the SSL server). Therefore, if the client trusts the issuer, it can be assured of the server's identity. Conversely, if the client does not know the issuer of the subject's certificate, then it can obtain the certificate of the issuer that issued the issuer's certificate. This process continues until the client obtains a certificate from a trusted issuer, or until a certificate is presented that was signed by the same entity presenting the certificate (called a self-signed certificate). In this circumstance, the subject of the certificate is vouching for itself.

When a self-signed certificate is presented to a client, it must determine whether to accept the certificate or not, and whether or not to allow the SSL session to be established. Many computer programs executing with user interfaces will typically force the human user to make the decision. The ZTP Network Security SSL Plug-In uses a callback routine to make this decision (see the Certificate Verification section on page 48).

After the client is satisfied with the server's identity, it uses the public key in the certificate to arrive at a shared secret between the client and the server, which in turn is used to encrypt all data transferred between them. This encryption occurs via the execution of a public key algorithm. If the server actually possesses the private key that corresponds to the public key in the certificate, then both the client and the server will arrive at the same secret. Otherwise, the secrets will not match, and data encrypted by one party will not be understood by the other party. The SSL handshake protocols are able to detect this condition and will immediately sever the SSL connection.

Certificate Chains

The previous section explained basic certificate concepts and indicated that certificate trust relationships are hierarchical in nature. Consequently, TLSv3 and SSLv1 servers are able to present the client with a list of certificates called a *certificate chain*. The first chain in the certificate belongs to the server presenting the list. The next certificate in the chain belongs to the issuer that signed the server's certificate. The next certificate belongs to the issuer that signed the issuer of the subject's certificate, etc.

SSL certificate chains can (but are not required to) end with a self-signed certificate (i.e., the one in which the certificate is issued to and issued by the same entity).

Note: SSLv2 servers are only permitted to supply the client with a single certificate, often a *self-signed* certificate. For the client to accept the server's certificate, it must trust one of the certificates in the server's chain. This trust can be developed as the result of having the previously stored issuer's certificate or simply prompting the user to accept the certificate.

When one of the SSL server handshake protocols in the ZTP Network Security SSL Plug-In is initialized (see the SSL Handshake Protocol Initialization section on page 22), the first parameter must be a pointer to the `CERT_CHAIN` data structure. This data structure accommodates a maximum of four X.509 certificates. The following code fragment shows an initialized variable of type `CERT_CHAIN` that contains a chain of two certificates:

```
CERT_CHAIN          CertChain =
{
  2,                  // 2 certificates in this chain
  BASE64_DER_ENCODED_DATA,// All certs & keys are in PEM format
  NULLPTR,            // Created by SSL layer when the chain is parsed
  {key_data, sizeof(key_data)-1},
                      // Private Key
  { {cert_data, sizeof(cert_data)-1},
                      // Subject Certificate
     {cert2_data, sizeof(cert2_data)-1},
                      // Issuer's Certificate
  {NULLPTR, 0},
  {NULLPTR, 0} }
};
```

The first member of the `CERT_CHAIN` structure indicates the number of certificates in the chain (must be between one and four, inclusive).

The second parameter specifies the encoding of all certificates and the subject's private key. Valid options are `BASE64_DER_ENCODED_DATA` (or, equivalently, `PEM_ENCODED_DATA`) or `DER_ENCODED_DATA`.

The third member of the `CERT_CHAIN` data structure contains a pointer to a data structure that the SSL protocol uses to parse information in the server's certificate. This parameter must always be set to `NULLPTR` when an application initializes a `CERT_CHIAIN` data structure.

The fourth member is a structure pointing to the first byte of the private key and to the length (in bytes) of the private key data.

The last member of the `CERT_CHAIN` is an array of (pointer, length) tuples that references the first byte of data in an X.509 certificate and the number of bytes of data in the X.509 certificate.

Generating Certificates

While developing an SSL-based application, either use one of the sample certificates and private key included in the `SSLDemo` folder, or create a new certificate and private key. To create a new certificate, it is necessary to obtain a third party tool. Zilog does not recommend or endorse any Certificate Authorities or third party utilities for the purpose of generating certificates and key pairs. However, for informational purposes, this section describes how to generate a certificate chain using OpenSSL (refer to www.openssl.org).

Note: The ZTP Network Security SSL Plug-In does not contain any utilities to generate public/private key pairs or generate X.509 certificate. Prior to putting a product into production, you should contact a certificate authority to request a signed X.509 certificate (and if required, a public/private key pair). Third party utilities can also be used to generate these items.

The first step is to generate a self-signed root certificate that will terminate the certificate chain. Generating this root certificate will generate a server certificate that will be signed with the root certificate, as the following instruction shows.

1. Generate a self-signed RSA (512-bit) certificate.

   In the following example, the interactive mode of the `OpenSSL req` command is used to enter information about the issuer. This information can also be provided in a configuration file.

```
OpenSSL> req -newkey rsa:512 -x509 -nodes -out Root.crt -keyout
RootKey.txt -set_serial 0x01 -days 365
Loading 'screen' into random state - done
Generating a 512 bit RSA private key
....++++++++++++
...++++++++++++
writing new private key to 'RootKey.txt'
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (e.g., city) []:San Jose
Organization Name (e.g., company) [Internet Widgets Pty
Ltd]:Zilog
Organizational Unit Name (e.g., section) []:.
Common Name (e.g., YOUR name) []:ZTP SSL CA
Email Address []:.
```

   This command generates a 512-bit RSA self-signed certificate with the subject and issuer common name set to `ZTP SSL CA`. The certificate will be valid for 365 days starting from the current date, and the certificate's serial number will be set to `01`. The X.509 certificate that is generated will be in PEM (i.e., Base64 ASN.1 DER data) and stored in a file named `Root.crt`. A PEM-encoded RSA private key file will also be generated and stored in a file named `RootKey.txt`. The `-nodes` option directs the `req` command not to DES-encrypt the private key. The Base64-encoded data in these files will be enclosed between the beginning and ending delimiter lines.

An example is shown in the following code fragment.

```
-----BEGIN CERTIFICATE-----
MIICQjCCAeygAwIBAgIBATANBgkqhkiG9w0BAQQFADBSMQswCQYDVQQGEwJVUzEL
MAkGA1UECBMCQ0ExETAPBgNVBAcTCFNhbiBKb3NlMQ4wDAYDVQQKEwVaaUxPRzET
MBEGA1UEAxMKWlRQIFNTTCBDQTAeFw0wNTEwMTUyMDAxMTZaFw0wNjEwMTUyMDAx
MTZaMFIxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJDQTERMA8GA1UEBxMIU2FuIEpv
c2UxDjAMBgNVBAoTBVppTE9HMRMwEQYDVQQDEwpaVFAgU1NMIENBMFwwDQYJKoZI
hvcNAQEBBQADSwAwSAJBALwIx2kBFRWBu7f17d4V+qe/By+6FGOzPus0rRtXEFPy
+M+11NISOLikREZV948QKN1GkT/8fJplhuMNn5G1LhsCAwEAAaOBrDCBqTAdBgNV
HQ4EFgQUHZCNWFT6S8lrh1+jSuTKIsZVk/8wegYDVR0jBHMwcYAUHZCNWFT6S8lr
h1+jSuTKIsZVk/+hVqRUMFIxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJDQTERMA8G
A1UEBxMIU2FuIEpvc2UxDjAMBgNVBAoTBVppTE9HMRMwEQYDVQQDEwpaVFAgU1NM
IENBggEBMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEEBQADQQALzgTerl+vD04M
PirnJIWeXpk3stLJ+yXhtVUp/puVRMx/cUNuK+B/Fko0MBJgWp8ILHf31DcHzXGQ
rpS8d8XM
-----END CERTIFICATE-----
```

2. Create the SSL server's certificate. The `OpenSSL req` command will be used again; however, this time a command file is used to supply the `req` command with information about the SSL server. This command will also generate a self-signed certificate for the server but, in the next step, the certificate will be signed by the root certificate created in the previous step.

```
OpenSSL> req -newkey rsa:512 -x509 -nodes -out SrvrSS.crt -keyout
SrvrKey.txt -config info.txt

Loading 'screen' into random state - done
Generating a 512 bit RSA private key
......................++++++++++++
..++++++++++++
writing new private key to 'SrvrKey.txt'
-----
OpenSSL>
```

The contents of the configuration file `info.txt` is shown in the following code fragment.

```
[ req ]
 distinguished_name      = req_distinguished_name
 prompt                  = no

 [ req_distinguished_name ]
```

```
C                       = US
ST                      = CA
L                       = San Jose
O                       = Zilog Inc.
CN                      = My SSL Server
emailAddress            = name@mycompany.com

[ req_attributes ]
```

3. Use the self-signed certificate created in Step 1 (the mock certificate authority's certificate is used in this example), to sign the certificate generated in Step 2. This authorization is accomplished using the `OpenSSL x509` command, as shown in the following code fragment.

   ```
   OpenSSL> x509 -days 100 -CA Root.crt -CAkey RootKey.txt -in
   SrvrSS.crt -out Srvr.crt -set_serial 0x1234
   Loading 'screen' into random state - done
   Signature ok
   subject=/C=US/ST=CA/L=San Jose/O=Zilog Inc./CN=My SSL Server/
   emailAddress=name@mycompany.com
   Getting CA Private Key
   OpenSSL>
   An example of the contents of the generated Srvr.crt text file
   (PEM encoded) follows:
   -----BEGIN CERTIFICATE-----
   MIIBvzCCAWmgAwIBAgICEjQwDQYJKoZIhvcNAQEEBQAwUjELMAkGA1UEBhMCVVMx
   CzAJBgNVBAgTAkNBMREwDwYDVQQHEwhTYW4gSm9zZTEOMAwGA1UEChMFWmlMT0cx
   EzARBgNVBAMTClpUUCBTU0wgQ0EwHhcNMDUxMDE1MjE1NDI1WhcNMDYwMTIzMjE1
   NDI1WjB9MQswCQYDVQQGEwJVUzELMAkGA1UECBMCQ0ExETAPBgNVBAcTCFNhbiBK
   b3NlMRMwEQYDVQQKEwpaaUxPRyBJbmMuMRYwFAYDVQQDEw1NeSBTU0wgU2VydmVy
   MSEwHwYJKoZIhvcNAQkBFhJuYW1lQG15Y29tcGFueS5jb20wXDANBgkqhkiG9w0B
   AQEFAANLADBIAkEAlHUuaMSbWu5jNAWDC8zTvM5JYQAsiJrXBkXDdOQKKHH6dlsH
   MmUFdpxNYXQQMUmhpsc4ktWQORqdEROXQYV16QIDAQABMA0GCSqGSIb3DQEBBAUA
   A0EAlnY7md2V/vKPWb/hNN9qbj/ZNNLGv+8QCii/3vm0WYsZFprS31FEcWDKu1aE
   r/2NIm4yqJfXFM4jJxaYThX7xg==
   -----END CERTIFICATE-----
   ```

   To use this file, cut and paste the text between the `BEGIN` and `END` delimiters and place it into a C array that will be referenced by a variable of type `CERT_CHAIN`. It is necessary to append line continuation characters (\) to each line of text, as shown in Step 4.

4. Create a `CERT_CHAIN` data structure which contains the server's certificate, the CA's root certificates and the server's private key. The following declaration shows an example.

   ```
   #include "SSL.h"
   ```

```
SSL_BYTE SrvrCrt[] = {"\
MIIBvzCCAWmgAwIBAgICEjQwDQYJKoZIhvcNAQEEBQAwUjELMAkGA1UEBhMCVVMx\
CzAJBgNVBAgTAkNBMREwDwYDVQQHEwhTYW4gSm9zZTEOMAwGA1UEChMFWmlMT0cx\
EzARBgNVBAMTClpUUCBTU0wgQ0EwHhcNMDUxMDE1MjE1NDI1WhcNMDYwMTIzMjE1\
NDI1WjB9MQswCQYDVQQGEwJVUzELMAkGA1UECBMCQ0ExETAPBgNVBAcTCFNhbiBK\
b3NlMRMwEQYDVQQKEwpaaUxPRyBJbmMuMRYwFAYDVQQDEw1NeSBTU0wgU2VydmVy\
MSEwHwYJKoZIhvcNAQkBFhJuYW1lQG15Y29tcGFueS5jb20wXDANBgkqhkiG9w0B\
AQEFAANLADBIAkEAlHUuaMSbWu5jNAWDC8zTvM5JYQAsiJrXBkXDdOQKKHH6dlsH\
MmUFdpxNYXQQMUmhpsc4ktWQORqdEROXQYV16QIDAQABMA0GCSqGSIb3DQEBBAUA\
A0EAlnY7md2V/vKPWb/hNN9qbj/ZNNLGv+8QCii/3vm0WYsZFprS31FEcWDKu1aE\
r/2NIm4yqJfXFM4jJxaYThX7xg=="};
SSL_BYTE RootCrt[] = {"\
MIICQjCCAeygAwIBAgIBATANBgkqhkiG9w0BAQQFADBSMQswCQYDVQQGEwJVUzEL\
MAkGA1UECBMCQ0ExETAPBgNVBAcTCFNhbiBKb3NlMQ4wDAYDVQQKEwVaaUxPRzET\
MBEGA1UEAxMKWlRQIFNTTCBDQTAeFw0wNTEwMTUyMDAxMTZaFw0wNjEwMTUyMDAx\
MTZaMFIxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJDQTERMA8GA1UEBxMIU2FuIEpv\
c2UxDjAMBgNVBAoTBVppTE9HMRMwEQYDVQQDEwpaVFAgU1NMIENBMFwwDQYJKoZI\
hvcNAQEBBQADSwAwSAJBALwIx2kBFRWBu7f17d4V+qe/By+6FGOzPus0rRtXEFPy\
+M+11NISOLikREZV948QKN1GkT/8fJplhuMNn5G1LhsCAwEAAaOBrDCBqTAdBgNV\
HQ4EFgQUHZCNWFT6S8lrh1+jSuTKIsZVk/8wegYDVR0jBHMwcYAUHZCNWFT6S8lr\
h1+jSuTKIsZVk/+hVqRUMFIxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJDQTERMA8G\
A1UEBxMIU2FuIEpvc2UxDjAMBgNVBAoTBVppTE9HMRMwEQYDVQQDEwpaVFAgU1NM\
IENBggEBMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEEBQADQQALzgTerl+vD04M\
PirnJIWeXpk3stLJ+yXhtVUp/puVRMx/cUNuK+B/Fko0MBJgWp8ILHf31DcHzXGQ\
rpS8d8XM"};

SSL_BYTE PrivKey[] = {"\
MIIBOgIBAAJBAJR1LmjEm1ruYzQFgwvM07zOSWEALIia1wZFw3TkCihx+nZbBzJl\
BXacTWF0EDFJoabHOJLVkDkanRETl0GFdekCAwEAAQJAPkM/KZV7ipF8ba76HRLU\
otTplZMbGlfGYs0Tgoy5behsGp+BG5mJJuS60CgzfFoxIFU5M18a+0njumsMogWG\
4QIhAMNglptCehhck24Bj0SsOD38xf4DwuX3lgr7hTC2VlVtAiEAwoWhStgbBAwH\
GwyjIsRBhy6haZHO7nxC6tLTIHmd4O0CIF+NeZrtZDFN9XyznpIDeG44lcypokQ+\
Vk+Au58bThXxAiASKvryi5aSXTE4tIh0EdJw9sj6nDSwj4iMeB5h9RnqzQIhAK1i\
ZFUHOYjTBanXj/Pl9QqNXqPn42u4B1GqOUJG/gmD"};

CERT_CHAIN   CertChain =
{
 2,                       // 2 certificates in this chain
 PEM_ENCODED_DATA,        // All certs & keys are in PEM format
 NULLPTR,                 // Created by SSL layer when the chain
                          // is parsed.
 {PrivKey, sizeof(PrivKey)-1},
                          // Server's Private Key
 { {SrvrCrt, sizeof(SrvrCrt)-1},
                          // Server's Certificate
   {RootCrt, sizeof(RootCrt)-1},
                          // Issuer's Certificate
```

```
  {NULLPTR, 0},
  {NULLPTR, 0} }
};
```

5. Lastly, to initialize the SSL server, use the certificate chain created in Step 4. For example, to use this certificate chain with the TLSv1 handshake protocol, use the following function call:

   ```
   TLS1_ServerInit( &CertChain, &DheParams );
   ```

Certificate Creation Issues

Consider the following points when creating your own certificates and private keys to be used with the SSL:

- The SSLv2 protocol always uses the RSA algorithm to exchange the Master Key during the establishment of a session. Therefore, X.509 certificates created for use with the SSLv2 protocol must contain an RSA Public Key, and the corresponding private key must be an RSA Private Key. Similarly, the constructed SSLv2 certificate chain must contain only one X.509 certificate.
- It is important to choose a key length that is appropriate for the importance of the data being exchanged. The sample certificates in the `Certificate` directory of the `SSLDemo` folder use a 512-bit public key. The longer the key, the less likely an attacker is to discover or hack the key. However, as key size increases, the SSL layer takes more time to complete the key exchange algorithm during the establishment of a session.
- The SSL layer in ZTP requires the private key to be in clear text format. Be sure that the utility used to generate the private key does not encrypt the output. To prevent encrypting of the private key, the `–nodes` option is used in the OpenSSL example, discussed earlier in this chapter. If the Private Key is encrypted, then the SSL layer will be unable to complete the key exchange, and it will not establish an SSL session.
- The X.509 certificate and Private Key must be encoded in the same manner. The SSL layer in ZTP cannot process these parameters if one is `DER_ENCODED_DATA` and the other is `BASE64_DER_ENCODED_DATA`.
- If the SSL server's Private Key and X.509 certificate are in the PEM format (`BASE64_DER_ENCODED_DATA`), they must be stored in RAM because the algorithm which converts PEM-formatted data into DER-formatted data (`DER_ENCODED_DATA`) performs the conversion in place (i.e., Base64 decoding overwrites the encoded data).
- Because private keys are stored in memory and must be transferred to the CPU over the system data bus, some form of physical security is required to prevent an attacker from analyzing the system memory or snooping the data bus and obtaining the private key.

Certificate Verification

Prior to using X.509 certificates, SSL clients and servers will perform integrity checks on the certificate to determine if it is authentic. For ZTP Network Security SSL Plug-In servers, these checks occur during the SSL handshake protocol's initialization call (see the SSL Handshake Protocol Initialization section on page 22). For clients, these checks occur when the server's certificate chain is received during the establishment of a session.

In the ZTP Network Security SSL Plug-In implementation, the following items are verified for each certificate in the chain:

- X.509 certificate structure
- The certificate's validity period is checked
- Certificate Signature (can be disabled)
- If the certificate is self-signed

By default, if the certificate contains all of the expected fields, is presented within its validity period, its signature has been verified and the certificate is not self-signed, the ZTP Network Security SSL Plug-In will implicitly trust the certificate. If any of these checks fail, a user-modifiable callback function is called. This callback function is named `VerifyCertificate`, and the default implementation (as shown in the following code fragment) is present in the `Certificate.c` configuration file.

```
SSL_STATUS VerifyCertificate
(
  SSL_X509_S         * pCertificate
)
{
  return( SSL_SUCCESS );
}
```

The purpose of this callback routine is to allow an application to examine information regarding a suspect certificate. If the `VerifyCertificate` callback returns `SSL_SUCCESS`, the certificate will be trusted and used to complete the establishment of a session. If the callback function returns `SSL_FAILURE`, the certificate will not be trusted; this situation will prevent an SSL session from being established. The default implementation simply accepts all suspect certificates.

The `flags` member of the `SSL_X509_S` structure referenced by the `pCertificate` pointer contains a combination of one or more of the following values which indicate the results of processing the certificate:

```
#define SSL_X509_PARSED_OK          0x01
#define SSL_X509_DATE_VALID         0x02
```

```
#define SSL_X509_SIGNATURE_VERIFIED 0x04
#define SSL_X509_SELF_SIGNED        0x08
#define SSL_X509_PERMANENT          0x10
#define SSL_X509_UNKNOWN_SIG_ALG    0x40
#define SSL_X509_TRUSTED            0x80
```

In general, certificates for which the `SSL_X509_PARSED_OK` flag is not set must never be trusted.

If the `SSL_X509_DATE_VALID` flag is not set, an attempt is made to use the certificate before or after its stated validity period. However, it could also be the case that the system date has not been set correctly.

The `SSL_SIGNATURE_VERIFIED` flag indicates if the SSL library is able to verify the signature on the certificate. This verification is possible with a self-signed certificate, but it will only be possible with other certificates if the SSL layer is in possession of the issuer's certificate (i.e., public key). Again, this verification is possible for all certificates in the certificate chain except, perhaps, for the last certificate. If the last certificate in the chain is self-signed, then its signature can be verified; if it is not, then your application must determine if the issuer should be trusted.

The `SSL_X509_SELF_SIGNED` flag indicates that the subject and issuer of the suspect certificate are identical; i.e., an entity is vouching for itself. Because the SSL layer has no way of determining if such a certificate is truly trustworthy, such certificates are always passed to the `VerifyCertificate` callback routine. In some cases, the certificate should be accepted without any question. For example, if a self-signed certificate is installed for a ZTP Network Security SSL Plug-In server, the `VerifyCertificate` callback function will be called. Clearly, this certificate must be accepted, because it is the one owned by your application.

In those cases in which a remote SSL server presents a certificate chain that does not end in a self-signed root certificate, it must be assumed that the client application is already in possession of the trusted root certificate, or implicitly trusts the certificate's issuer. In all other cases, the certificate might not be trustworthy.

Generally, the only certificates that are marked `SSL_X509_PERMANENT` are the local server certificates. However, your application is permitted to set this flag on any certificate presented to the `VerifyCertificate` callback for which `SSL_SUCCESS` is being returned. This allowance will prevent the SSL layer from releasing resources associated with the certificate.

The `SSL_X509_UNKNOWN_SIG_ALG` flag indicates one possible reason why the *signature verified* flag is not set (i.e., if the ZTP Network Security SSL Plug-In does not implement or has been configured not to support the signature algorithm that the issuer used to sign the certificate), then it will not be possible to verify the signature. In this case, the `SSL_X509_UNKNOWN_SIG_ALG` flag will be set and the `SSL_X509_SIGNATURE_VERIFIED` flag will be cleared.

The SSL protocol layer internally sets the `SSL_X509_TRUSTED` flag on all certificates that it implicitly trusts, or on those certificates for which the `VerifyCertificate` callback function returns `SSL_SUCCESS`.

Note: The ZTP Network Security SSL Plug-In does not check if the certificate has been revoked. No attempt is made to contact certificate issuers and obtain a list of certificates that have been revoked. If this functionality is required, it must be implemented within the `VerifyCertificate` callback function.

Verifying All Certificates

In some instances, it is useful to examine all certificates – even those that the SSL layer implicitly trusts (i.e., the `SSL_X509_TRUSTED` flag is set), especially if the application must perform additional integrity checks on the certificate beyond the basic verification performed in the SSL library. For example, the application can obtain the issuer's *certificate revocation list* (CRL) to determine if an otherwise-valid certificate should be rejected.

To force the `VerifyCertificate` callback to be called for all certificates processed by the SSL handshake protocols, set the value of the `SSL_PresentAllCertificates` configuration variable to TRUE. This variable is located in the `ssl_conf.c` configuration file. Its default definition is shown in the following code fragment:

```
SSL_BOOL SSL_PresentAllCertificates = FALSE;
```

Signature Verification

A digital signature provides a mechanism that allows an entity to verify that another entity was the originator of a specific piece of digital information. This verification establishes the authenticity of the information. To generate a signature, the information is typically hashed into a fixed-sized quantity using a digest algorithm, and then subjected to a public key operation using the signatory's private key.

Any other entity possessing the signatory's public key can then perform the complementary public key operation using the signatory's public key and compare the recovered digest to a locally-generated digest of the same piece of information. If these digests are identical, the information is deemed authentic; i.e., the signatory's private key has not been compromised. If the key remains secure, then by the nature of the asymmetric public key algorithm, it is extremely unlikely that an attacker could forge the signature on an altered block of information.

All X.509 certificates used by SSL must be signed by an issuer. By verifying the signature on a certificate, the recipient can be relatively certain that the certificate is accurate and identical to the information that the issuer originally signed. If it can be proven (via an asymmetric key exchange/agreement algorithm) that the subject of the certificate is in possession of the private key corresponding to the public key in the certificate, then the certificate recipient can be relatively certain that it is communicating with the entity to which the certificate was issued.

When the SSL client and the server establish a session using Ephemeral Diffie-Hellman parameters (or temporary RSA keys), these parameters are also digitally signed by the SSL server. If the client verifies the signature on these parameters, it can be relatively certain that the parameters were created by the SSL server and not an attacker attempting to trick the client to use bogus parameters which the attacker can decode.

By default, the ZTP Network Security SSL Plug-In will attempt to verify all digital signatures. However, this verification can require the execution of many public key algorithms which take considerable CPU bandwidth. At the customer's discretion, verification of digital signatures can be disabled. The customer is advised that doing so will lower the overall security of the system. However, in applications requiring faster session establishment times, disabling the verification of digital signatures could be a viable option.

Disabling Signature Verification

Digital signature verification is controlled by the value of the `SSL_VerifySignatures` configuration variable located in the `ssl_conf.c` configuration file. The default setting is shown in the following code fragment:

```
SSL_BOOL SSL_VerifySignatures = TRUE;
```

Note: Disabling signature verification is useful only for SSL clients. SSL servers in the ZTP Network Security SSL Plug-In will always generate signatures when required, regardless of the setting of the `SSL_VerifySignatures` variable. In addition, because client authentication is not supported, SSL servers in this implementation will never verify a client signature.

Limitations

Because the ZTP Network Security SSL Plug-In only supports a limited set of cryptographic operations, it can only verify (and generate) digital signatures that use these supported algorithms. A digital signature requires the use of a digest algorithm and a public key signature algorithm. This implementation supports two digest algorithms (MD5 and SHA1) and two signature algorithms (RSA and DSA). Therefore, the only digital signature algorithms that can be supported are:

- MD5 with RSA encryption
- SHA1 with RSA encryption
- SHA1 with DSA

Note: The digital signature standard (DSS) specification does not permit the use of MD5 with DSA. Therefore, this implementation will not recognize MD5 with DSA as a valid signature.

Session Cache

Executing public key algorithms is a computationally-intensive process, and accounts for nearly all of the time required to establish an SSL session. As the length of the keys involved increases, execution time increases exponentially. To prevent the execution of these asymmetric algorithms each time a session is initiated, the same client and same server must establish a new session in all versions so that the SSL handshake protocol uses a session cache.

This session cache effectively stores the shared secret which a given client and server derive using a public key algorithm from a previous session. If both parties store this shared secret in the session cache, then the next time they attempt to establish a session, there will be no need to execute another public key algorithm to arrive at a common shared secret.

In the ZTP Network Security SSL Plug-In implementation, the session cache is controlled by the value of two configuration variables, `SSL_MAX_SESSION_CACHE_ENTRIES` and `SSL_CACHE_TIMEOUT`. These variables are defined in the `ssl_conf.c` configuration file. The default configuration is shown in the following code fragment:

```
SSL_BYTE             SL_MAX_SESSION_CACHE_ENTRIES = 8;
SSL_DWORD SSL_CACHE_TIMEOUT = 30000; /* measured in 10ms ticks */
```

`SSL_MAX_SESSION_CACHE_ENTRIES` determines the maximum number of entries in the cache. One entry is used for each SSL session established using a different remote IP address. This addressing requirement allows multiple remote sockets (i.e., individual connections) to share the same SSL session. For example, if an SSL session is established with remote socket 1.2.3.4:5000, then it creates a new entry in the session cache which will be reused if a connection is attempted with remote socket 1.2.3.4:6000. If the `SSL_MAX_SESSION_CACHE_ENTRIES` variable is set to 0, then the SSL session cache is disabled; i.e., all attempts at establishing an SSL session will be required to perform asymmetric public key operations to arrive at a new shared secret every time the same client and server reconnect.

The `SSL_CACHE_TIMEOUT` variable determines the maximum lifetime (measured in 1/100th of a second) of an idle entry in the session cache. In general, leaving entries in the session cache indefinitely is a security risk because the longer the shared secret remains in existence, the greater the likelihood that an attacker will be able to find it. Conversely, if session entries expire too fast, then extra public key operations must be performed, resulting in slow session establishment times. The default configuration specifies that an idle entry in the session cache will expire in five minutes.

Session Cache Operation

When a new session is established, the SSL protocol layers adds a new entry to the session cache and sets the entry's expiry timer to `SSL_CACHE_TIMEOUT`. As more sessions are established, additional entries in the cache are used. If the cache is full (i.e., it contains `SSL_MAX_SESSION_ENTRIES`) and a new session is established, the recently used entry will be evicted from the cache.

Each time the SSL protocol layers search the cache for a specific entry and that entry is found, its time out value is reset to `SSL_CACHE_TIMEOUT`. While searching the session cache, the SSL protocol layer will evict expired entries.

Diagnostic Messages

The ZTP Network Security SSL Plug-In is capable of generating a considerable amount of diagnostic information. This information is displayed on the console device when the handshake protocols execute. To control the amount of diagnostic messages displayed, the value of the `SSL_DEBUG_LEVEL` configuration variable can be modified; this variable is defined in the `ssl_conf.c` configuration file. The default configuration is shown in the following code fragment.

```
SSL_BYTE SSL_Debug_level = SSL_DEBUG_ERROR;
```

This variable can be set to any one of the four values listed in Table 7.

Table 7. Diagnostic Message Control

SSL_DEBUG_LEVEL Setting	Description
`SSL_DEBUG_NONE`	Suppress all diagnostics messages
`SSL_DEBUG_ERROR`	Display only Error messages
`SSL_DEBUG_WARNING`	Display only Error and Warning messages
`SSL_DEBUG_INFO`	Display all diagnostics messages

How to Use the HTTPS Server

The SSL libraries contain an HTTPS server that can serve encrypted web pages to client browsers. This HTTPS server is initialized by calling the `https_init` API. This API takes the same number and type of parameters as the standard HTTP server API.

For example, to initialize the standard HTTP server in ZTP, the following command is used:

```
http_init(http_defmethods,httpdefheaders,website,80);
```

To initialize the HTTPS server, the following command is used:

```
https_init(http_defmethods,httpdefheaders,website,443);
```

The HTTPS server will be accessible to clients using the same version of the SSL handshake protocol as the ZTP Network Security SSL Plug-In. Therefore, if the SSL version 2, SSL version 3, and TLS version 1 handshake protocols have all been initialized in server mode, the HTTPS server will be accessible to SSL clients using any of these protocol versions. For more information about initializing these different SSL protocols, see the SSL Handshake Protocol Initialization section on page 22 and the Client Mode or Server Mode Support section on page 24.

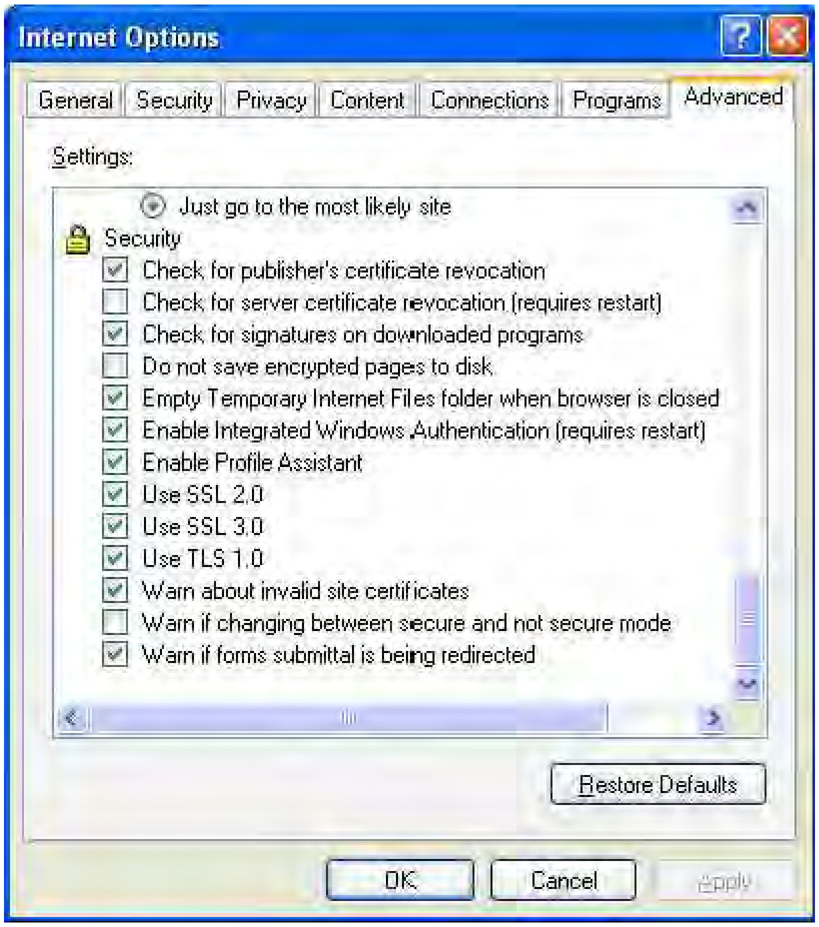
It is possible to have both secure and nonsecure web servers running at the same time. However, the two webservers must be on different ports. The port number used for nonsecure HTTP servers is 80; for secure HTTP servers (HTTP over SSL or HTTPS), use port 443.

For more information about the HTTP (or HTTPS) server, refer to the section headed *How to Use HTTP* in the ZTP Network Security SSL Plug-In Reference Manual (RM0047).

Limitations of the ZTP HTTPS Server

Consider the following points when using the ZTP HTTPS server.

1. It could become necessary to configure your client browser to support the same version of SSL that is used by the ZTP HTTPS server; consult the documentation for your browser for configuration assistance. Using Microsoft Internet Explorer as an example, navigate to **Tools** → **Internet Options**, click the **Advanced** tab, and ensure that the appropriate SSL protocols are selected. Figure 6 on page 55 displays this Advanced tab in the Internet Options dialog in Internet Explorer.



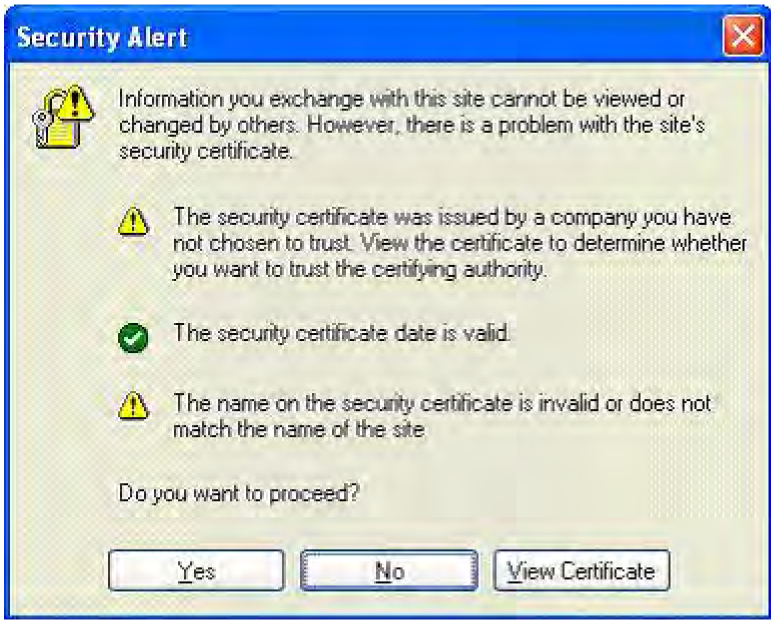
Just go to the most likely site
Security
- [✓] Check for publisher's certificate revocation
- [ ] Check for server certificate revocation (requires restart)
- [✓] Check for signatures on downloaded programs
- [ ] Do not save encrypted pages to disk
- [✓] Empty Temporary Internet Files folder when browser is closed
- [✓] Enable Integrated Windows Authentication (requires restart)
- [✓] Enable Profile Assistant
- [✓] Use SSL 2.0
- [✓] Use SSL 3.0
- [✓] Use TLS 1.0
- [✓] Warn about invalid site certificates
- [ ] Warn if changing between secure and not secure mode
- [✓] Warn if forms submittal is being redirected

Figure 6. Internet Options Window

Note: When multiple SSL protocols are enabled, preference is given to the higher protocol. Therefore, if both TLSv1 and SSLv2 are selected, the browser first connects using TLS and only reverts to SSLv2 if the TLS connection attempt fails.

2. When using the sample certificate with the HTTPS server in the SSL demo project, be aware that client browsers such as Microsoft Internet Explorer generate warning messages when processing the sample certificate (see Figure 7 on page 56). The first warning is encountered because the certificate was self-signed; therefore, a trusted root certificate authority does not exist in the certificate chain. The second warning is generated because the certificate's subject (the distinguished name, or identity of the SSL server) does not match the server's website or IP address. These warnings are not

generated when the CA issues a valid certificate in which the CN value matches the server's name or IP address.



Security Alert

Information you exchange with this site cannot be viewed or changed by others. However, there is a problem with the site's security certificate.

The security certificate was issued by a company you have not chosen to trust. View the certificate to determine whether you want to trust the certifying authority.

The security certificate date is valid.

The name on the security certificate is invalid or does not match the name of the site

Do you want to proceed?

Figure 7. Security Alert

3. All ZTP SSL servers share the same certificate. However, each ZTP SSL handshake protocol is configured to use different certificates. For more information about ZTP SSL server configuration, see the SSL Handshake Protocol Initialization section on page 22.

Creating SSL Applications

This chapter explains how to migrate a TCP-based client or server application to use SSL.

Note: UDP-based applications cannot use SSL.

Automatic Protocol Negotiation

When a remote SSL client attempts to establish an SSL session with a local server, the remote client is free to use any version of the SSL handshake protocol it requires (i.e., SSLv2, SSLv3 or TLSv1). If the corresponding SSL handshake protocol layer in the ZTP Network Security SSL Plug-In is also initialized and a compatible cipher suite is negotiated, then the session is established.

When a local ZTP SSL client attempts to establish a connection with a remote SSL server, the ZTP Network Security SSL Plug-In, on its initial attempt, will use the highest version of the SSL handshake protocol initialized in client mode. For example, if TLSv1 and SSLv3 and SSLv2 have all been initialized in client mode (see the Client Mode or Server Mode Support section on page 24), then the TLSv1 handshake protocol will first be used to attempt to establish the session. If this session fails, the ZTP SSL layer will automatically reattempt the connections using the next-highest version of the SSL handshake protocol layer initialized in client mode (SSLv3 in this example). This process continues until either the session is established or until all client-enabled handshake protocols fail to establish the session.

SSL Applications in ZTP-Based Systems

Transferring encrypted data using any of the ZTP SSL handshake protocols follows the same semantics as transferring data using the ZTP TCP layer. However, the syntax is slightly different.

Server Applications

This section provides a procedure that a ZTP TCP server process uses to create a TCP connection, and shows the modification required to use the SSL layer. For more information about the TCP socket APIs, refer to the *API Definitions* section of the ZTP Network Security SSL Plug-In Reference Manual (RM0047). Observe the following procedure to establish a TCP-SSL connection in server mode:

1. To open a TCP-SSL server socket, a TCP server application in ZTP must first create a TCP server socket that must be listening for the connections. The following code fragment offers an example.

```
INT16 sockfd;
INT16 confd;
struct sockaddr_in server;
struct sockaddr_in client;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
server.sin_addr.s_addr = INADDR_ANY;
server.sin_family = AF_INET;
server.sin_port = intel16(0x1234);

bind(sockfd, (struct sockaddr *)&server, sizeof(struct
sockaddr_in));
listen(sockfd, 1);
```

This API opens the TCP socket and requests a server socket to be created on TCP port `0x1234`. To create an SSL server application, the above code is modified, as shown in the following code fragment:

```
INT16 sockfd;
INT16 confd;
struct sockaddr_in server;
struct sockaddr_in client;

sockfd = socket(AF_INET, SOCK_SSL, 0);
server.sin_addr.s_addr = INADDR_ANY;
server.sin_family = AF_INET;
server.sin_port = intel16(0x1234);

bind(sockfd, (struct sockaddr *)&server, sizeof(struct
sockaddr_in));
listen(sockfd, 1);
```

To accept a TCP-SSL connection from a remote client, the code informs the TCP server to wait for an incoming TCP connection request, as shown in the following example:

```
struct sockaddr_in client;
confd = accept(sockfd, (struct sockaddr *)&client, (INT16
*)&addrlen);
```

The same API is used to wait for an incoming SSL connection request. When a connection is established, the SSL handshake is complete, and the `confd` variable is released to the application, which transfers the SSL data.

2. To receive TCP data over the TCP-SSL connection, the `recv` API is used. For example, to receive 10 bytes of TCP data and place the data in a buffer called `MyBuff`, the following code fragment can be used:

   ```
   BYTE MyBuff[100];
   INT16 Status;
   Status = recv( ConnectionDev, MyBuf, 10, 0);
   ```

   The exact same application is used to receive 10 bytes of data through the SSL layer.

Note: Although the data sent between the client and server SSL layers is encrypted, the data passed between the ZTP SSL layer and the user application is nonencrypted. Therefore, the code that retrieves data from the ZTP TCP layer can also be used to retrieve decrypted data from the ZTP SSL layer without modification.

   To send TCP data, the `send` API is used. For example, to send 10 bytes of TCP data from a buffer called `MyBuff`, the following code fragment is used:

   ```
   Status = send( ConnectionDev, MyBuf, 10, 0);
   ```

   The exact same application is used to send 10 bytes through the SSL layer.

3. To close an underlying TCP-SSL connection, the `close_s` API is used, with the socket of the connection device (used during data transfer) passed as a parameter.

   ```
   close_s( ConnectionDev );
   ```

   The exact same `close_s` API is also used to close the SSL session represented by the SSL connection socket. When it is no longer necessary to maintain the TCP server in a running condition, the application can close the TCP socket by issuing the `close_s` API and using the TCP socket ID.

   ```
   close_s( ServerDev );
   ```

   Again, the exact same `close_s` API is also used to close the SSL server device.

In summary, any ZTP TCP server application is converted to use SSL for secure data transfer by changing the socket type used. The syntax and semantics of all other data transfer APIs are identical for both TCP and SSL.

Client Applications

This section presents steps that a ZTP TCP client process uses to create a TCP connection, and shows the modification required to use the SSL layer.

Observe the following procedure to establish a TCP-SSL connection in client mode:

1. To open a TCP/SSL client, a TCP client application in ZTP uses the `connect()` API to request a connection to a specific remote server. For example:

```
INT16 sockfd;
UINT32 dstipaddr;

struct sockaddr_in server;
struct sockaddr_in client;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
dstipaddr = name2ip("172.16.6.204");
server.sin_addr.s_addr = intel(dstipaddr);
server.sin_family = AF_INET;
server.sin_port = intel16(0x1234);

connect(sockfd, (struct sockaddr *)&server, sizeof(struct
sockaddr) );
```

This API opens the TCP socket and requests a TCP connection to port `0x1234` on the remote device on which the IP address is 172.16.6.204. If the connection is successfully established, `sockfd` will reference the TCP socket dedicated to this connection. If the connection fails, `connect()` returns a negative value. To create an SSL connection socket in client mode, the `connect()` call is modified, as shown in the following code fragment:

```
INT16 sockfd;
UINT32 dstipaddr;
struct sockaddr_in server;
struct sockaddr_in client;

sockfd = socket(AF_INET, SOCK_SSL, 0);

dstipaddr = name2ip("172.16.6.204");

server.sin_addr.s_addr = intel(dstipaddr);
server.sin_family = AF_INET;
server.sin_port = intel16(0x1234);
connect(sockfd, (struct sockaddr *)&server, sizeof(struct
sockaddr) );
```

This API opens the SSL socket (SSL) and requests an SSL connection to port `0x1234` on the remote device on which the IP address is 172.16.6.204. If the SSL session is successfully established, `socketfd` will reference the SSL device driver ID dedicated to this connection. If an SSL session cannot be established, `connect()` returns a negative value.

2. To receive TCP data over the TCP-SSL connection, the `recv` API is used. For example, to receive 10 bytes of TCP data and place the data in a buffer called `MyBuff`, the following code fragment can be used:

```
BYTE MyBuff[100];
INT16 Status;
Status = recv( sockfd, MyBuf, 10, 0);
```

The exact same API is used to receive 10 bytes of data through the SSL layer.

Note: Although the data sent between the client and server SSL layers is encrypted, the data passed between the ZTP SSL layer and user application is nonencrypted. Therefore, the code that retrieves data from the ZTP TCP layer can also be used to retrieve decrypted data from the ZTP SSL layer without modification.

To send TCP data, the `send` API is used. For example, to send 10 bytes of TCP data from a buffer called `MyBuff`, the following code fragment can be used:

```
Status = send( ConnectionDev, MyBuf, 10, 0);
```

This exact same API is also be used to send 10 bytes through the SSL layer.

3. To close an underlying TCP/SSL connection, the `close_s()` API is used with the socket of the connection (used during the data transfer) that is passed as a parameter.

```
close_s( sockfd );
```

The exact same `close_s` API is also used to close the SSL session represented by the SSL connection socket.

In summary, any ZTP TCP client application is converted to use SSL for secure data transfer by changing the socket type used from `SOCK_STREAM` to `SOCK_SSL`. The syntax and semantics of all other data transfer APIs are identical for both TCP and SSL.

For more information about the TCP socket layer APIs, refer to the *API Definitions* chapter of the ZTP Network Security SSL Plug-In Reference Manual (RM0047).

Appendix A. Default SSL Cipher Suites

This appendix identifies the subset of the cipher suites defined in the SSL version 2, SSL version 3 and TLS version 1 specifications, which are supported by the ZTP Network Security SSL Plug-In.

In general, an SSL cipher suite is comprised of the following components:

- A key exchange algorithm used to establish a shared secret between the client and server
- A cipher algorithm used for encrypting and decrypting data through the SSL layer
- A digest algorithm (known as a *hash*) used to compute a Message Authentication Code which allows the recipient of an SSL data record to verify that the data sent by the peer was not altered in transit

By using various combinations of algorithms for these components, a large number of cipher suites can be supported, subject to the implementation limitations discussed in this appendix.

SSL Version 2 Cipher Suites

The SSL Version 2 specification limits the choice of key exchange algorithm and hash function to RSA and MD5, respectively. Therefore, the SSL2 cipher suite is determined by the choice of cipher algorithm (and corresponding symmetric key size). Because this implementation does not support the RC2 or IDEA ciphers, cipher suites using these algorithms cannot be supported.

Table 8 shows the cipher suites defined in the SSL Version 2 specification, and indicates which are supported by the ZTP Network Security SSL Plug-In.

Table 8. SSLv2 Cipher Suites

Cipher Suite Mnemonic	Supported?
SSL_CK_RC4_128_WITH_MD5	Yes
SSL_CK_RC4_128_EXPORT40_WITH_MD5	Yes
SSL_CK_RC2_128_CBC_WITH_MD5	No
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	No
SSL_CK_IDEA_128_CBC_WITH_MD5	No
SSL_CK_DES_64_CBC_WITH_MD5	Yes
SSL_CK_DES_192_CBC_WITH_MD5	Yes

Note: When SSLv2 was drafted, the U.S. export laws restricted the length of the encryption keys to 40 bits and public keys to 512 bits. Therefore, when the longer keys are exchanged only 40 bits of the key can be encrypted. The remaining key must be sent in clear text. Similarly, the public key size used in export cipher suites must be restricted to 512 bits or less.

SSL Version 3 Cipher Suites

The SSLv3 and TLSv1 cipher suites contained in their respective specifications are nearly identical. The only significant difference is the SSLv3 specification included support for the Fortezza key exchange algorithm, which is not included in the TLSv1 specification. Otherwise, the only difference between the cipher suites is all SSLv3 cipher suites use *SSL* as the first three characters in the cipher suite mnemonic; while TLSv1 cipher suites use *TLS*. Therefore, the SSLv3 cipher suite `SSL_RSA_WITH_RC4_128_MD5` is identical to the TLSv1 cipher SUITE `TLS_RSA_WITH_RC4_128_MD5`.

Table 9 shows the cipher suites defined in the SSL version 3 specification and indicates which of them are supported by the ZTP Network Security SSL Plug-In.

Table 9. SSLv3 Cipher Suites

Cipher Suite Mnemonic	Supported?
SSL_RSA_WITH_NULL_MD5	Yes
SSL_RSA_WITH_NULL_SHA	Yes
SSL_RSA_EXPORT_WITH_RC4_40_MD5	Yes
SSL_RSA_WITH_RC4_128_MD5	Yes
SSL_RSA_WITH_RC4_128_SHA	Yes
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	No
SSL_RSA_WITH_IDEA_CBC_SHA	No
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
SSL_RSA_WITH_DES_CBC_SHA	Yes
SSL_RSA_WITH_3DES_EDE_CBC_SHA	Yes
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	Yes
SSL_DH_DSS_WITH_DES_CBC_SHA	Yes
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	Yes
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
SSL_DH_RSA_WITH_DES_CBC_SHA	Yes
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	Yes

Table 9. SSLv3 Cipher Suites (Continued)

Cipher Suite Mnemonic	Supported?
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	Yes
SSL_DHE_DSS_WITH_DES_CBC_SHA	Yes
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	Yes
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
SSL_DHE_RSA_WITH_DES_CBC_SHA	Yes
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	Yes
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	No
SSL_DH_anon_WITH_RC4_128_MD5	No
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	No
SSL_DH_anon_WITH_DES_CBC_SHA	No
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	No
SSL_FORTEZZA_KEA_WITH_NULL_SHA	No
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	No
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	No

Note: When SSLv3 was drafted, the U.S. export laws restricted the length of the encryption keys to 40 bits and public keys to 512 bits. Therefore, when cipher algorithms are used which require longer key lengths, only 40 bits of the key are protected by the key exchange algorithm. Similarly, the public key size used in export cipher suites must be restricted to 512 bits or less. The public keys used for signature verification are not restricted in export cipher suites, but the key size of the (Ephemeral) Diffie-Hellman parameters must be 512 bits or less.

TLS Version 1 Cipher Suites

The SSLv3 and TLSv1 cipher suites contained in their respective specifications are nearly identical. The only significant difference is that the SSLv3 specification included support for the Fortezza key exchange algorithm, which is not included in the TLSv1 specification. Otherwise, the only difference between the cipher suites is all SSLv3 cipher suites use *SSL* as the first three characters in the cipher suite mnemonic, while TLSv1 cipher suites use *TLS*. Therefore, the SSLv3 cipher suite: `SSL_RSA_WITH_RC4_128_MD5` is identical to the TLSv1 cipher SUITE `TLS_RSA_WITH_RC4_128_MD5`.

Table 10 shows the cipher suites defined in the TLS version 1 specification, and indicates which are supported by the ZTP Network Security SSL Plug-In.

Table 10. TLSv1 Cipher Suites

Cipher Suite Mnemonic	Supported?
TLS_RSA_WITH_NULL_MD5	Yes
TLS_RSA_WITH_NULL_SHA	Yes
TLS_RSA_EXPORT_WITH_RC4_40_MD5	Yes
TLS_RSA_WITH_RC4_128_MD5	Yes
TLS_RSA_WITH_RC4_128_SHA	Yes
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	No
TLS_RSA_WITH_IDEA_CBC_SHA	No
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
TLS_RSA_WITH_DES_CBC_SHA	Yes
TLS_RSA_WITH_3DES_EDE_CBC_SHA	Yes
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	Yes
TLS_DH_DSS_WITH_DES_CBC_SHA	Yes
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	Yes
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
TLS_DH_RSA_WITH_DES_CBC_SHA	Yes
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	Yes
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	Yes
TLS_DHE_DSS_WITH_DES_CBC_SHA	Yes
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	Yes
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	Yes
TLS_DHE_RSA_WITH_DES_CBC_SHA	Yes
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	Yes
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	No
TLS_DH_anon_WITH_RC4_128_MD5	No
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	No
TLS_DH_anon_WITH_DES_CBC_SHA	No
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	No

Note: When SSLv3 was drafted, the U.S. export laws restricted the length of the encryption keys to 40 bits, and public keys to 512 bits. Therefore, when cipher algorithms are used that require longer key lengths, only 40 bits of the key are protected by the key exchange algorithm. Similarly, the public key size used in export cipher suites must be restricted to 512 bits or less. The public keys used for signature verification are not restricted in export cipher suites, but the key size of the (Ephemeral) Diffie-Hellman parameters must be 512 bits or less.

AES Extensions

The advanced encryption standard (AES) is being adopted because the U.S. government prefers symmetric ciphers; it is intended to replace the older data encryption standard (DES). Because the SSL specifications were drafted prior to the standardization of AES, they do not define any AES-based cipher suites. RFC 3268 defines a set of cipher suites compatible with the TLSv1 specification.

Table 11 shows the AES-based cipher suites defined in RFC 3268, and indicates which are supported by the ZTP Network Security SSL Plug-In.

Table 11. SSLv2 Cipher Suites

Cipher Suite Mnemonic	Supported?
TLS_RSA_WITH_AES_128_CBC_SHA	Yes
TLS_DH_DSS_WITH_AES_128_CBC_SHA	Yes
TLS_DH_RSA_WITH_AES_128_CBC_SHA	Yes
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	Yes
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	Yes
TLS_DH_anon_WITH_AES_128_CBC_SHA	No
TLS_RSA_WITH_AES_256_CBC_SHA	Yes
TLS_DH_DSS_WITH_AES_256_CBC_SHA	Yes
TLS_DH_RSA_WITH_AES_256_CBC_SHA	Yes
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	Yes
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	Yes
TLS_DH_anon_WITH_AES_256_CBC_SHA	No

Appendix B. Advanced Topic: Creating Private Cipher Suites

When the SSL specifications were originally drafted, they contained a default set of supported cipher suites. A cipher suite is a combination of PKI algorithm, symmetric cipher, and digest algorithm used to secure data exchanged in an SSL session. The specifications also permitted implementors to define their own cipher suites. This feature is useful only in environments in which the implementor has control over the code used by both clients and servers, because third party implementations are unlikely to recognize the implementor's cipher suites. In addition, the implementor must ensure that the codes used to define their cipher suites are unique in their environment. If an implementor defines a new cipher suite code (for example, `0xFF`, `0x7C`), then this code must be understood by all SSL devices in the environment (i.e., the same PKI, cipher, and digest algorithms), or else it will not be possible to establish SSL sessions.

Users of the ZTP Network Security SSL Plug-In international distribution are only permitted to define new cipher suites that are a combination of cryptographic algorithms which are currently supported. If you are using the U.S. version, modify the source code to the cryptographic library to add additional algorithms that can be used to define new cipher suites.

This section provides a simple example to show how to add a new TLSv1 cipher suite.

RFC 3268 defines a number of standard cipher suites that can be added to the TLS protocol to support AES. Some of these cipher suites are already supported in this implementation. All of the cipher suites specified in RFC 3268 use the SHA1 digest algorithm. In this example, a private AES-based cipher suite is defined for the ZTP Network Security SSL Plug-In, which uses MD5.

Procedure

1. Examine the cipher suite codes defined in the `CipherSuite.h` header file, as shown in the following code strings.

   ```
   #define TLS_RSA_WITH_RC4_128_MD5  0x0400
   #define TLS_RSA_WITH_RC4_128_SHA  0x0500
   ```

   Notice that the last byte of these code strings is `0x00`. Private cipher suites must use a value of `0xFF` in the cipher suite code. Therefore, the `0x11FF` value is used for the new cipher suite.

   For this cipher suite, it is appropriate to use the RSA, AES and MD5 algorithms; therefore, a suitable mnemonic for the cipher suite is:

   `PRIVATE_RSA_WITH_AES_128_CBC_MD5`

This mnemonic indicates that RSA will be used for authentication and key exchange; 128-bit AES will be used as the symmetric cipher, and MD5 will be used as the digest algorithm. Therefore, add the following definition to the `CipherSuite.h` header file:

```
#define PRIVATE_RSA_WITH_AES_128_CBC_MD5 0x11FF
```

2. A new entry must be created in the cipher suite table for each of the SSL handshake protocols in which this cipher suite must be supported. The definition of the cipher suite entry is shown in the following code fragment:

```
{
 PRIVATE_RSA_WITH_AES_128_CBC_MD5,
 SSL_PKI_RSA,
 SSL_CIPHER_AES,
 SSL_HASH_MD5,
 FALSE,
 AES_128_KEY_SIZE_BYTES,
 AES_IV_SIZE_BYTES,
 MD5_HASH_SIZE_BYTES,
 TRUE
},
```

For the cipher suite to take effect, it is necessary to rebuild the project. To give this cipher suite preference, place it immediately after the definition of the NULL cipher suite. Alternatively, all other cipher suites in the table can be removed or can be disabled by setting the last entry in the other cipher suites to FALSE. For more information about this topic, see the Cipher Suite Configuration section on page 31.

Appendix C. Diffie-Hellman Private Keys

Although this implementation supports the establishment of an SSL session using Diffie-Hellman certificates with the SSLv3 and TLSv1 handshake protocols, it must be noted that this use of Diffie-Hellman certificates is extremely rare. Consequently, few utilities are able to generate Diffie-Hellman certificates, and even fewer utilities will generate a Diffie-Hellman private key. Those utilities that do output DH private keys are likely to do so in different formats. The PKCS#3: Diffie-Hellman Key Agreement Standard does not specify the format of the DH private key.

Therefore, this implementation uses the simplest possible encoding of the DH private key consistent with the ASN.1 definition in PKCS#15 – a single ASN.1 DER-encoded integer containing the value of the private key:

```
DHPrivateKey ::= INTEGER -- private key, x
```

As an example, the DER encoding of the private key `0x12345678` is:

```
02 04 12 34 56 78
```

The segments in this key can be defined as:

`02`:	ASN.1 INTEGER.
`04`:	Length of the integer in octets.
`12 34 56 78`:	Value of the integer MSB first.

Customer Support

To share comments, get your technical questions answered or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.