AN4683
Application note

Transport layer security protocol for SPWFxxx module

Introduction

The purpose of this document is to present a demonstration package for creating a secure connection over TCP/IP between the Wi-Fi module SPWF01Sxxx (see [1] in *References*)and a remote server exposing secured service.

Security is provided by the secure sockets layer (SSL) and transport layer security (TLS) protocols. The SSL/TLS protocols provide security for communication over networks, such as the Internet, and allow client and server applications to communicate in a way that is confidential and secure.

The document includes a brief introduction to SSL/TLS principles, a description of the demonstration package organization and a tutorial with client-server connection examples.

Contents

1 SSL/TLS protocol overview

Originally developed by Netscape in the mid 1990s, the secure sockets layer (SSL) is a cryptographic protocol designed to provide communication security over the Internet (see [3] in *References*). Version 1.0 never publicly released, while version 2.0 was released in February of 1995 but “contained a number of security flaws which ultimately led to the design of SSL version 3.0” (see [4] in *References*). SSLv3.0 was a complete redesign of the protocol and is still widely supported (since 1996).

The IETF standards body adopted SSLv3.0 with minor tweaks and published it as TLS version 1.0 (RFC 2246, 1999). The two versions are very similar, but interoperability is precluded. TLSv1.2 (RFC 5246, 2008) is the latest and recommended version, which is superior because it offers flexibility and key features that were unavailable in earlier protocol versions.

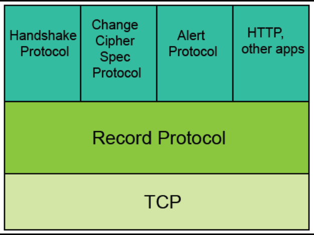
All TLS versions were further refined (RFC 6176, 2011) removing their backward compatibility with SSL such that TLS sessions will never negotiate the use of SSLv2.0.

As shown in *Figure 1*, SSL/TLS is typically applied in TCP/IP protocol stacks and provides security services on top of the transport layer. The protocol is composed of two layers: the TLS record layer and the TLS handshake layer.

At the lowest level, layered on top of a reliable transport protocol is the TLS record protocol. The record protocol is used for encapsulation of various higher-level protocols and provides two basic properties:

- Confidentiality
- Integrity

Figure 1. SSL/TLS protocol architecture



The TLS Handshake layer consists of three sub-protocols: Handshake, Change cipher spec, and Alert.

The Handshake protocol is the most complex part of TLS and provides a number of very important security functions. It allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. The TLS handshake protocol provides connection security that has three basic properties:

- Cipher suite negotiation
- Authentication of the server and, optionally, of the client
- Session key information exchange

An outline of the sub-protocols is provided in the next section.

1.1 SSL/TLS sub-protocols

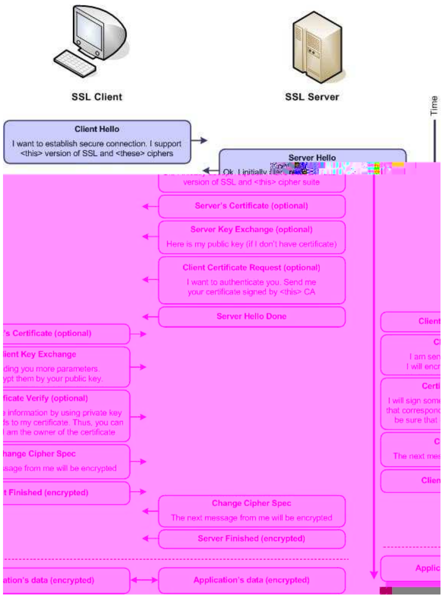
The TLS record layer and TLS handshake layer consists of four sub-protocols overall: handshake protocol, change cipher spec protocol, alert protocol, and record protocol.

1.1.1 Handshake protocol

This sub-protocol is used to negotiate session information between the client and the server. The session information consists of a session ID, peer certificate(s), the cipher suite, the compression algorithm, and a shared secret that is used to generate the session key.

Figure 2 depicts the message flow for a full handshake process (see [5] in *References*). The optional value indicates optional or situation-dependent messages, i.e. in the case of mutual authentication a TLS server must send its certificate and request a certificate from the client, while in the case of anonymous negotiation the optional messages may be skipped. The certificate-based authentication is examined more in detail in *Section 1.2*.

Figure 2. SSL/TLS full handshake procedure


SSL Client
SSL Server
Time
Client Hello
I want to establish secure connection. I support <this> version of SSL and <these> ciphers
Server Hello
Ok. I initially
version of SSL and <this> cipher suite
Server's Certificate (optional)
Server Key Exchange (optional)
Here is my public key (if I don't have certificate)
Client Certificate Request (optional)
I want to authenticate you. Send me your certificate signed by <this> CA
Server Hello Done
's Certificate (optional)
lient Key Exchange
ding you more parameters.
ypt them by your public key.
ficate Verify (optional)
information by using private key
ds to my certificate. Thus, you can
I am the owner of the certificate
hange Cipher Spec
ssage from me will be encrypted
t Finished (encrypted)
Change Cipher Spec
The next message from me will be encrypted
Server Finished (encrypted)
ation's data (encrypted)
Application's data (encrypted)

1. The client sends a ClientHello message specifying the highest SSL/TLS protocol version (SSLv3.0, TLSv1.0, 1.1 or 1.2) it supports, a random number, a list of suggested cipher suites and compression methods.
2. The server responds with a ServerHello message, containing the chosen protocol version, another random number, cipher suite and compression method from the choices offered by the client, and the session ID.

Note: The chosen protocol version should be the highest that both client and server support.

Note: The client and the server must support at least one common cipher suite, otherwise the handshake protocol fails. The server generally chooses the strongest common cipher suite they both support.

3. The server sends its digital certificate in an optional certificate message. For example, the server uses X.509 digital certificates.
4. Additionally, a ServerKeyExchange message may be sent, if it is required (e.g., if the server has no certificate, or if its certificate is for signing only).
5. If the server requires a digital certificate for client authentication, an optional CertificateRequest message is appended.
6. The server sends a ServerHelloDone message indicating the end of this phase of negotiation.
7. If the server has sent a CertificateRequest message, the client must send the Certificate message. For example the client uses an X.509 digital certificate.
8. The client sends a ClientKeyExchange message. This message contains the premaster secret used in the generation of the symmetric encryption keys and the message authentication code (MAC) keys. The client encrypts the pre-master secret with the public key of the server.

Note: The public key is sent by the server in the digital certificate or in ServerKeyExchange message.

9. If the client sent a digital certificate to the server, the client sends a CertificateVerify message signed with the client's private key. By verifying the signature of this message, the server can explicitly verify the ownership of the client digital certificate.
10. The client sends a ChangeCipherSpec message announcing that the new parameters (cipher method, keys) have been loaded.
11. The client sends a finished message. It is the first message encrypted with the new cipher method and keys.
12. The server responds with a ChangeCipherSpec and a finished message from its end.
13. The SSL handshake protocol ends and the encrypted exchange of application data can be started.

During the initial handshaking phase, the client and server negotiate cipher suites, which specify a cipher for each of the following functionalities:

Table 1. Ciphers

Functionality	Cipher
Authentication	RSA, DSA, ECDSA
Key-exchange/agreement	RSA, DH, ECDH, SRP, PSK
Symmetric ciphers for encryption	RC4, IDEA, DES, 3DES, AES or Camellia.
Hash	MAC (for SSLv3.0) or HMAC with MD2, MD4, MD5, SHA-1, SHA-256 (after TLSv1.1 and 1.2 standards).

A complete list of SSL/TLS cipher suites can be found in the registry maintained by the Internet assigned numbers authority (IANA) (see [6] in *References*).

1.1.2 Change cipher spec protocol

The change cipher spec protocol is used to change the keying material used for encryption between the client and server. Keying material is raw data that is used to create keys for cryptographic use. The change cipher spec protocol consists of a single message to tell the other party in the SSL/TLS session that the sender wants to change to a new set of keys. The key is computed from the information exchanged by the handshake protocol.

1.1.3 Alert protocol

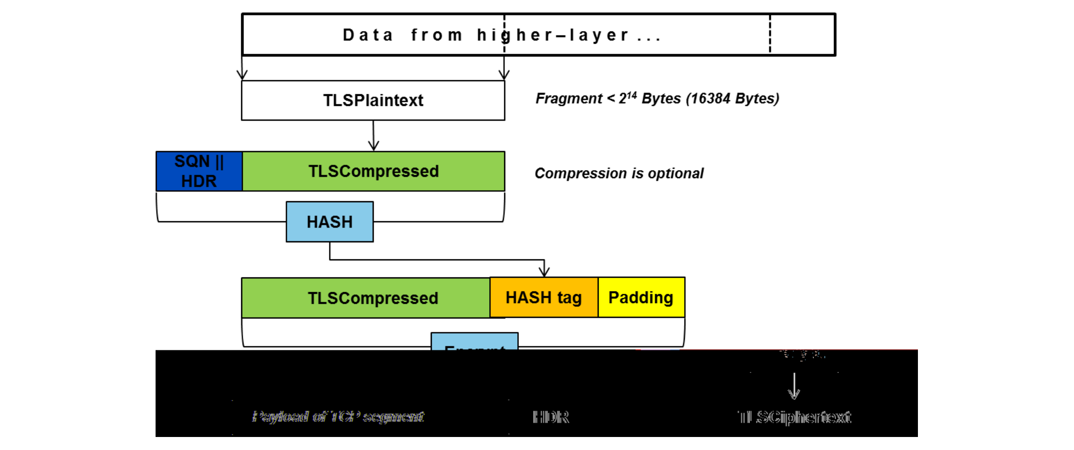
Alert messages are used to indicate a change in status or an error condition to the peer. There are a wide variety of alerts to notify the peer of both normal and error conditions. Alerts are commonly sent when the connection is closed, an invalid message is received, a message cannot be decrypted, or the user cancels the operation.

1.1.4 Record protocol

The record protocol receives and encrypts data from the higher-layer and delivers it to the transport layer. As shown in *Figure 3*, the record protocol takes the data, fragments it into TLSPlaintext blocks with a size appropriate to the cryptographic algorithm. Then it optionally compresses (or, for data received, decompresses) the TLSPlaintext, applies a MAC or HMAC (HMAC is supported only by TLS) to get the hash tag. Finally the TLSCompressed data and hash tag (and some padding eventually) are concatenated and encrypted (or decrypted) using the information negotiated during the handshake protocol.

Encryption and hash ensure, respectively, the confidentiality and the integrity of the plaintext.

Figure 3. Record protocol operations



[redacted]

1.2 Authentication and certificates

SSL/TLS requires a server certificate and, optionally, a client certificate. The digital certificate certifies the ownership of a public key by the named subject of the certificate, also known as public key certificates. This allows others parties to rely upon signatures or assertions made by the private key that corresponds to the public key that is certified.

Digital certificates used in SSL/TLS comply with the X.509 standard (see [8] in *References*), which specifies the information required and the formats for public key certificates. In an

X.509 system, the subject of the certificate is identified by a distinguished name (DN). A DN is a series of name-value pairs that uniquely identify an entity. The following attribute types are commonly found in the DN:

CN	Common name
T	Title
O	Organization name
OU	Organization unit name
L	Locality name
ST (or SP or S)	State or province name
C	Country

The X.509 standard provides for a DN to be specified in a string format. For example:

CN=John, O=STM, OU=Test, C=IT

The common name (CN) can describe an individual user or any other entity, for example a web server. DNs may include a variety of other name-value pairs. The rules governing the construction of DNs can be complex. For comprehensive information about DNs (see [8] in *References*).

In this model of trust relationships, a certification authority (CA) is an independent and trusted third party that issues digital certificates to provide assurance that the public key of an entity truly belongs to that entity. The roles of a CA are:

- On receiving a request for a digital certificate, to verify the identity of the requester before building, signing and returning the personal certificate
- To provide the CA's own public key in its CA certificate
- To publish lists of certificates that are no longer trusted in a certificate revocation List (CRL).

An X.509 certificate issued by the CA binds a particular public key to the name of the DN the certificate identifies. Only the public key certified by the certificate will work with the corresponding private key possessed by the DN identified by the certificate.

The contents of a certificate, according to the X.509 version 3 specifications, may include:

- The version number of the X.509 standard supported by the certificate.
- The certificate’s serial number. Every certificate issued by a CA has a serial number that is unique among the certificates issued by that CA.
- Information about the user’s public key, including the algorithm used and a representation of the key itself.
- The DN of the CA that issued the certificate.
- The period during which the certificate is valid.
- The DN of the certificate subject, which is also called the subject name. For example, in an SSL client certificate, this is the user’s DN.
- Optional certificate extensions, which may provide additional data used by the client or server.
- The cryptographic algorithm, or cipher, used by the issuing CA to create its own digital signature.
- The CA’s digital signature, obtained by hashing all of the data in the certificate together and encrypting it with the CA’s private key.

The certificate text format begins with the following line:

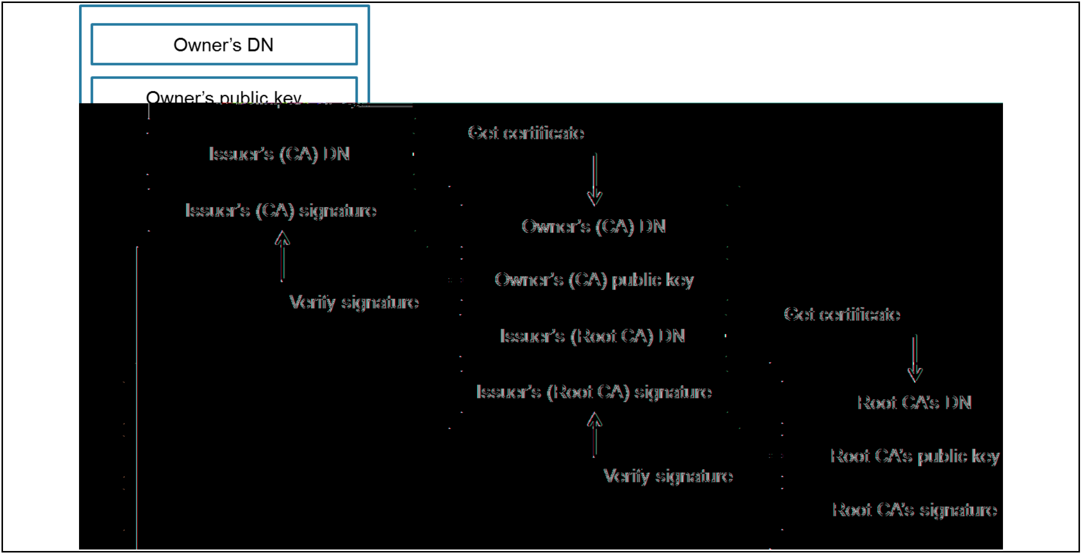
-----*BEGIN CERTIFICATE*-----

followed by certificate data, which should be base-64 encoded, as described by RFC 1113. The certificate information must end with this last line:

-----*END CERTIFICATE*-----

When you receive the certificate from another entity, you may need to use a certificate chain, also known as the certification path, which is a list of certificates used to authenticate an entity. The chain, or path, begins with the certificate of that entity, and each certificate in the chain is signed by the entity identified by the next certificate in the chain. The chain terminates with a root CA certificate. The root CA certificate is always signed by the CA itself; it must be considered as a trusted CA and must be available in the application (e.g. SSL/TLS client, web browser). The signatures of all certificates in the chain must be verified until the root CA certificate is reached. *Figure 4* illustrates a certification path from the certificate owner to the root CA, where the chain of trust begins. Notice that different chains can have multiple or even none intermediate CAs.

Figure 4. Certificate chain or chain of trust



In some cases it would be easier and less expensive to use self-signed certificates, for example, for testing purposes or when the parties know and trust each other. A self-signed certificate is a certificate that is signed by the same entity whose identity it certifies. There is no central CA.

TLS supports three authentication modes:

- **Mutual authentication** - both parties (client and server) share their signed certificates and authenticate each other. Mutual authentication provides stronger security by assuring that the identity on both sides of the communication are known.
- **One-way authentication** - only the server sends its signed certificate and is authenticated by the client. The client is not required to send the server a digital certificate and remains unauthenticated (no certificate).
- **Anonymous** - neither entity authenticates the identity of the other party.

Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked. In case of one-way or mutual authentication, because certificate validation requires that root CA keys be distributed independently, it is assumed the remote end must already possess a root CA certificate to accomplish validation.

2 SPWF01Sxxx use modes

The demonstrator allows secure TCP/IP connection to be created between the Wi-Fi module SPWF01Sxxx (see [1] in *References*) and a remote server exposing secured service. The SPWF01Sxxx module includes a lightweight SSL/TLS stack and a cryptographic library targeted for embedded applications. The following sections illustrate the main security features supported.

2.1 TLS protocol

The SPWF01Sxxx module integrates a lightweight SSL/TLS stack and a cryptographic library. To meet device memory constraints, the demonstrator enables just a subset of cryptographic algorithms with respect to *Table 1*. The SPWF01Sxxx implements a SSL/TLS client with the features listed below:

- SSLv3.0 and TLSv1.0, 1.1 and 1.2, with automatic downgrade of protocol version
- Server and client authentication
- Multiple hashing functions: MD5, SHA-1, SHA-256
- Block, stream, and authenticated ciphers: AES (128 and 256, CBC), 3DES, ARC4
- Public key algorithms: RSA (1024, 2048), ECDSA
- Key exchange: ECDH, ECDHE
- ECC support: EC curves over 192, 224, 384, 256, 521 bit prime field
- X.509 certificate support, PEM format

2.2 Supported ciphers list

Table 2. Demonstrator cipher suites

Cipher suites	2 byte codes (hex format)	Version
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x27	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	0xC0,0x23	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x29	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	0xC0,0x25	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	0xC0,0x0A	TLS1.0/1.1/1.2
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	0xC0,0x05	TLS1.0/1.1/1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	0xC0,0x09	TLS1.0/1.1/1.2
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	0xC0,0x04	TLS1.0/1.1/1.2
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	0xC0,0x07	TLS1.0/1.1/1.2
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	0xC0,0x02	TLS1.0/1.1/1.2
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x08	TLS1.0/1.1/1.2
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x03	TLS1.0/1.1/1.2

Table 2. Demonstrator cipher suites (continued)

Cipher suites	2 byte codes (hex format)	Version
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	0xC0,0x14	TLS1.0/1.1/1.2
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	0xC0,0x0F	TLS1.0/1.1/1.2
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	0xC0,0x13	TLS1.0/1.1/1.2
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	0xC0,0x0E	TLS1.0/1.1/1.2
TLS_ECDHE_RSA_WITH_RC4_128_SHA	0xC0,0x11	TLS1.0/1.1/1.2
TLS_ECDH_RSA_WITH_RC4_128_SHA	0xC0,0x0C	TLS1.0/1.1/1.2
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x12	TLS1.0/1.1/1.2
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x0D	TLS1.0/1.1/1.2
TLS_RSA_WITH_AES_256_CBC_SHA256	0x00,0x3D	TLS1.2
TLS_RSA_WITH_AES_128_CBC_SHA256	0x00,0x3C	TLS1.2
TLS_RSA_WITH_AES_256_CBC_SHA	0x00,0x35	TLS1.0/1.1/1.2
TLS_RSA_WITH_AES_128_CBC_SHA	0x00,0x2F	TLS1.0/1.1/1.2
SSL_RSA_WITH_RC4_128_SHA	0x00,0x05	SSL3.0/TLS1.0/1.1/1.2
SSL_RSA_WITH_RC4_128_MD5	0x00,0x04	SSL3.0/TLS1.0/1.1/1.2
SSL_RSA_WITH_3DES_EDE_CBC_SHA	0x00,0x0A	SSL3.0/TLS1.0/1.1/1.2

2.3 Domain name check for server certificate

When making a TLS connection, the client requests a digital certificate from the server. Once the server sends the certificate, the client examines it and compares the domain it was trying to connect to with the name (common name or others) included in the certificate. If a match is found, the connection proceeds as normal. If a match is not found, the user may be warned of the discrepancy and the connection may be aborted as the mismatch may indicate an attempted man-in-the-middle attack.

The demonstrator allows the user to bypass the warning to proceed with the connection, with the user taking on the responsibility of trusting the certificate and the connection.

2.4 Authentication

As mentioned in *Section 1*, SSL/TLS requires a server certificate and, optionally, a client certificate to be exchanged during handshake. Depending on the required level of trust, we can have three authentication modes: anonymous, one-way and mutual authentication. The following three sub-sections describe how to configure the SPWF01Sxxx module to enable the three different modes.

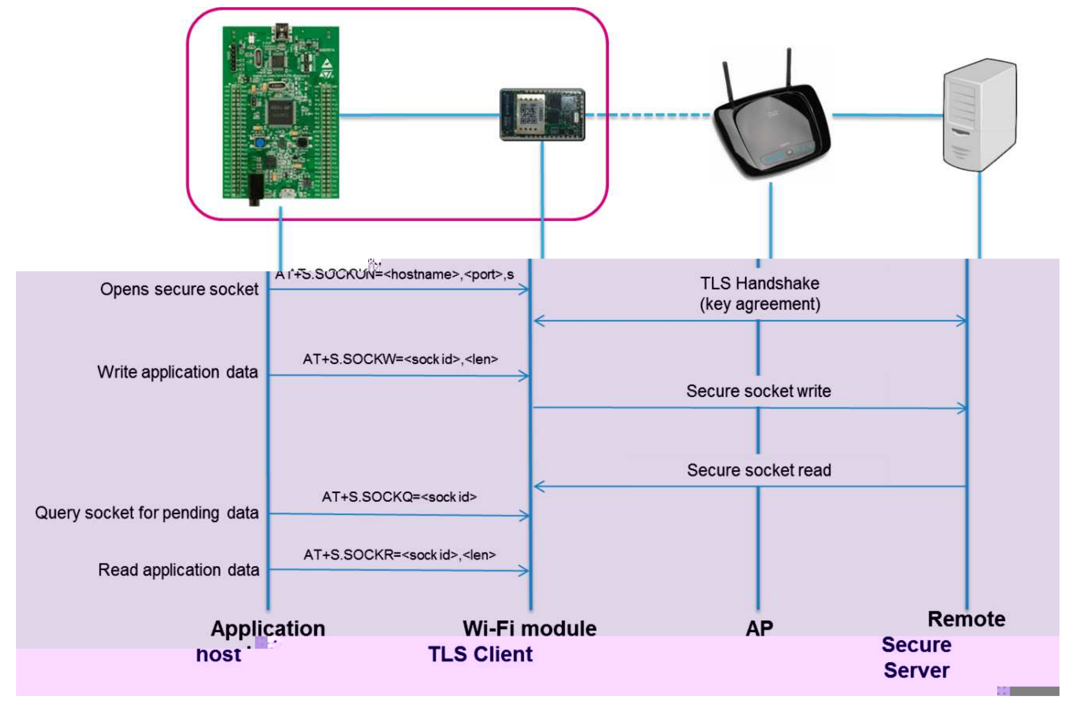
2.4.1 Anonymous negotiation

In case of anonymous TLS connection, the user assumes to be in a trusted environment, thus party authentication is not required. Even if server sends a certificate, the client will skip the verification of authenticity.

Note: This mode is particularly attractive for privacy-preserving solutions and also for low memory consumption, since it does not require certificates storage. But it increases the chance for intruder-in-the-middle attacks.

Figure 5 lists the AT required commands for opening a secure connection with anonymous negotiation, followed by writing to and reading sockets.

Figure 5. AT commands for anonymous negotiation



2.4.2 One-way authentication

In one-way authentication mode, the server sends its signed certificate to an unauthenticated client.

In order to verify the server certificate, the client:

1. verifies the digital signature
2. checks that the date of the certificate is in range
3. verifies the domain

To achieve this purpose, the following steps must be completed:

1. the issuing root CA certificate must be loaded in advance into client (see AT+S.TLSCERT) in PEM format.
2. To check the date, the module reference time must be initialized after each module reset (see AT+S.SETTIME): the time refers to UTC format and must be expressed as the time in seconds since 1970-Jan-01.
3. The domain passed to the client (see AT+S.TLSDOMAIN) must match the name specified in the server certificate (Common Name or others).

If the verification of the server certificate succeeds, the connection proceeds as normal.

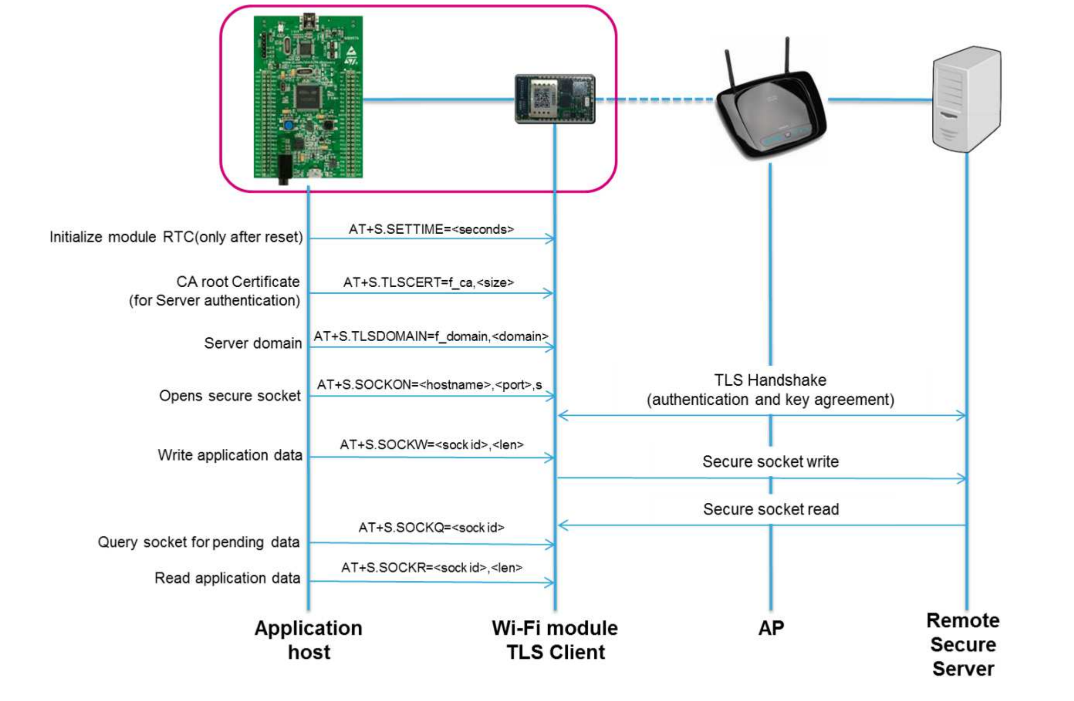
If verification fails (either for signature verification error, time or domain mismatch), then the client throws a warning message “ERROR: SSL/TLS unable to connect” and the connection is closed. As such, to solve this connection error, users either need to turn off verification of the server (i.e switch to anonymous mode; see *Section 2.4.1*), or load the correct CA certificate.

Note: The one-way authentication works with any cipher reported in Table 2, including all ECC ciphers up to 521 key length, RSA-1024 and RSA-2048.

Note: The maximum allowed size for files uploaded to module is approximately 1.3 KB.

After the client is configured as in *Figure 6*, the host can open a secure socket.

Figure 6. AT commands for one-way authentication



2.4.3 Mutual authentication

In mutual authentication mode, both parties (client and server) share their signed certificates.

In order to verify server certificate, the client:

1. verifies the digital signature
2. checks that the date of the certificate is in range
3. verifies the domain

Additionally, for client authentication, the client also:

4. sends its certificate to the server. Be aware that, in order to verify the client certificate, the server should have access to the issuing CA certificate (public or private).

To achieve this purpose, the following steps must be completed:

1. The issuing Root CA certificate must be loaded in advance into client (see AT+S.TLSCERT) in PEM format.
2. To check the date, the module reference time must be initialized after each module reset (see AT+S.SETTIME): the time refers to UTC format and must be expressed as the time in seconds since 1970-Jan-01.
3. The domain passed to the client (see AT+S.TLSDOMAIN) must match the name specified in the server certificate (Common Name or others).
4. The certificate and private key of the client must be loaded in advance into client (see AT+S.TLSCERT) in PEM format.

The server is in charge of client authentication. If the client authentication fails, it's up to the server to either keep or close the connection:

- Option 1: If client authentication succeeds, the handshake is completed and the connection proceeds as normal.
- Option 2: if the client authentication fails but the server neglects it, then the handshake is completed and the connection proceeds as normal.
- Option 3: if the client authentication fails and this blocks the server, then the handshake is interrupted, the connection is reset by the server and the client generates a warning message “ERROR: SSL/TLS unable to connect” (the connection is closed).

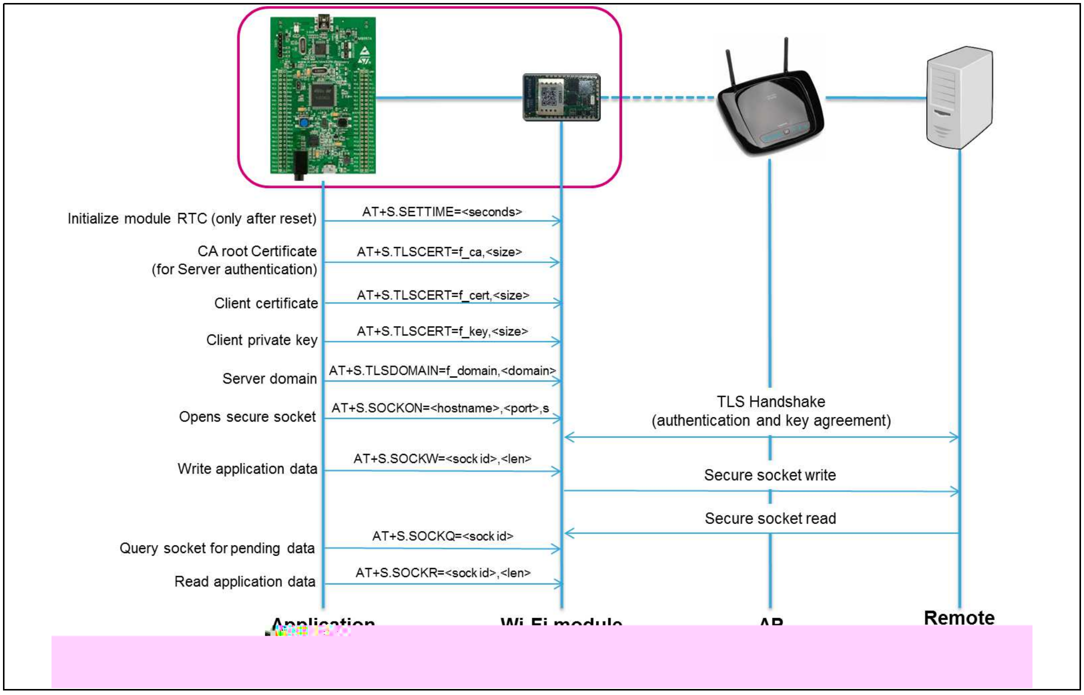
As such, to solve connection errors, users either need to change authentication mode, i.e. switch to anonymous mode (*Section 2.4.1*) or one-way mode (*Section 2.4.2*), or load the correct certificates and key.

Note: The mutual mode is clearly more resource demanding than previous ones. Due to memory limitations, the adoption of mutual authentication is limited to specific authentication ciphers: it is recommended to use ECC ciphers up to 521 key length, while RSA-1024 and RSA-2048 should be avoided as it could exceed memory resources.

The maximum allowed size for files uploaded to the module (CA certificate, domain name, client certificate and client key) is approximately 3 KB overall.

After the client is configured as in *Figure 7*, the host can open a secure socket.

Figure 7. AT commands for mutual authentication


Initialize module RTC (only after reset)
AT+S.SETTIME=<seconds>
CA root Certificate
(for Server authentication)
AT+S.TLSCERT=f_ca,<size>
Client certificate
AT+S.TLSCERT=f_cert,<size>
Client private key
AT+S.TLSCERT=f_key,<size>
Server domain
AT+S.TLSDOMAIN=f_domain,<domain>
Opens secure socket
AT+S.SOCKON=<hostname>,<port>,s
TLS Handshake
(authentication and key agreement)
Write application data
AT+S.SOCKW=<sock id>,<len>
Secure socket write
Secure socket read
Query socket for pending data
AT+S.SOCKQ=<sock id>
Read application data
AT+S.SOCKR=<sock id>,<len>
Application
Wi-Fi module
AP
Remote

2.5 Protocol version downgrade

The client is able to connect to a server running SSLv3.0 to TLSv1.2 protocol version. Thanks to the version downgrade capability, the client can use the highest protocol version supported by the server and eventually downgrade to SSLv3.0 if needed.

2.6 Pseudo random number generator

The SPWF01Sxxx module provides a software-based pseudo random number generator (PRNG). The PRNG is initialized at boot time and is based on the RC4 stream cipher algorithm.

3 Demonstration package

The functions of the demonstration package are summarized in *Table 3*.

Table 3. Demonstration package functions

SSL/TLS Client	Available versions: SSLv3.0 and TLSv1.0, 1.1 and 1.2. Automatic downgrade of protocol version.
Cipher suites	*Table 2*
– Public key algorithms	RSA (1024, 2048), ECDSA
– Key-exchange/agreement	ECDH, ECDHE
– Symmetric ciphers for encryption	AES (128 and 256, CBC), 3DES, ARC4
– Hash	MD5, SHA-1, SHA-256
– ECC support	EC curves over 192, 224, 384, 256, 521 bit prime field
Certificates	X.509 certificate support, PEM format.
Configurations	Anonymous For module configuration, see *Section 2.4.1* and *Figure 5*.
	One-way authentication For module configuration, see *Section 2.4.2* and *Figure 6*. In particular notice that: – Module reference time must be initialized – Root CA certificate and domain name of the server must be loaded. – Authentication ciphers: ECDSA with all EC curves, RSA-1024 and RSA-2048. – Maximum allowed size for files uploaded to module is approximately 1.3KB.
	Mutual authentication For module configuration, see *Section 2.4.3* and *Figure 7*. In particular: – Module reference time must be initialized – Root CA certificate and domain name of the server, client certificate and client key files must be loaded. – Authentication ciphers: ECDSA with all EC curves. – Maximum allowed size for files uploaded to module is approximately 3KB overall.

3.1 Package directories

The demonstration package is contained in the project folder and is organized as described below:

- Project/firmware: this folder contains the firmware binary

Project/examples: contained here are the certificates and configuration files for test examples.

The examples folder contains four test examples for creating a TLS secure connection between the SPWF01Sxxx module (client) and a remote server:

- **Example 1:** TLS Client with Mutual authentication. The client, server and CA use ECC authentication (*Section 3.3*).
- **Example 2:** TLS Client with one-way authentication. The server and CA use RSA-2048 authentication (*Section 3.2*)
- **Example 3:** Gmail SMTP server access with Anonymous negotiation (*Section 3.5*).
- **Example 4:** Xively example with Anonymous negotiation (*Section 3.6*)

3.2 Wi-Fi module setup

This section describes the essential steps for setting up the SPWF01Sxxx module to enable the TLS features. A complete list of AT commands, including a detailed description of the use of the commands, are contained in (see [2] in *References*).

To couple the SPWF01Sxxx module with a Wi-Fi access point (SSID, WPA-PSK passphrase) providing a connection to the Local Area Network (LAN) and to the Internet, these are the necessary AT commands:

```
AT+S.SCFG=wifi_mode,0
AT+S.SCFG=wifi_priv_mode,2
AT+S.SSIDTXT=<SSID>
AT+S.SCFG=wifi_wpa_psk_text,<WPA PSK passphrase>
AT+S.SCFG=wifi_mode,1
AT&W
AT+CFUN=1
```

Once the module is connected to the access point, it is ready to open a secure socket.

Notice that, due to memory limitations, the SPWF01Sxxx module allows the opening of one single secure socket at a time; i.e. if you want to open a new secure socket, you must first close any open one.

3.3 Example 1: TLS Client with mutual authentication

This first example implements a TLS connection supporting mutual authentication. The client, server and CA use EC cipher.

Before running this example, the SPWF01Sxxx (client) must be connected to the Wi-Fi LAN as illustrated in *Section 3.2* and OpenSSL must be installed on the PC (acting as server). For testing purposes, OpenSSL-1.0.1i has been used as TLS Server. Documentation and installation instructions are available on the OpenSSL website (see [12] in *References*).

The IP addresses of the client and server are automatically assigned by the network router.

To start the server, open a command prompt in the folder Project/Examples/Example1 and run this line:

```
openssl s_server -cert server_cert.pem -key server_key.pem -CAfile
root_ca_of_client.pem -Verify 2 -verify_return_error -accept <port>
```

Note: *The accept parameter specifies the TCP port to listen for connections. If not specified, the default 4433 is used.*

To start the client, it is recommended to run AT+S.TLSCERT2=clean,all to clean the Flash memory, and then use these AT commands:

```
AT+S.SETTIME=<seconds>
AT+S.TLSCERT=f_ca,<size><CR><data>
AT+S.TLSCERT=f_cert,<size><CR><data>
AT+S.TLSCERT=f_key,<size><CR><data>
AT+S.TLSDOMAIN=f_domain,<server domain>
AT+S.SOCKON=<hostname>,<port>,s[,ind]
```

Note: *Seconds is the current time expressed in seconds since 1970-01-01: AT+S.SETTIME can be done only once after module reset.*

AT+S.TLSCERT=<f_ca/f_cert/f_key>,<size><CR><data> stores certificates or key files (in PEM format) to Flash memory of the module. The first input parameter is used to indicate when a root CA certificate (f_ca), a client certificate (f_cert) or key file (f_key) is passed to the module. This command accepts data after the <CR> at the end of the command line. The host is expected to supply <size> of data as last parameter of the command line. The size values must be expressed in bytes.

AT+S.TLSDOMAIN=f_domain,<server domain> passes the server domain to the module and stores the domain information to Flash memory. The server domain is a string of characters that must match the name specified in the server certificate (Common Name or others).

When using AT+S.TLSCERT and AT+S.TLSDOMAIN with f_ca, f_cert, f_key and f_domain, the file information is stored to Flash memory: since Flash memory is non-volatile, any file stored in Flash memory will remain intact even when switching off or resetting the module. Thus, to remove file information from Flash you can use the command AT+S.TLSCERT2=clean,<f_ca|f_cert|f_key|f_domain> to selectively remove a specific file, or AT+S.TLSCERT2=clean,all to remove all files information from Flash ones.

Note that loading files to Flash memory is always preferred! Alternatively, you may use AT+S.TLSCERT=<ca/cert/key>,<size><CR><data> and AT+S.TLSDOMAIN=domain,<server domain> to load files to RAM, but note that (a) RAM is volatile and in this case the loaded file information is lost after switch off or module reset; (b) files stored in RAM have higher priority than Flash ones; (c) files upload to RAM is deprecated in favor of Flash.

AT+S.SOCKON=<hostname>,<port>,s[,ind] opens a secure socket to server hostname on port. The hostname is the IP address of the server, the port must correspond to the one specified in OpenSSL server options, and the s parameter specifies a request for secure socket. The parameter ind is optional and provides the indication (WIND:55, see below) that some data has been received: if enabled, it is strongly suggested to immediately empty the buffer when a pending data is received (see SOCKQ and SOCKR commands below).

When the TLS handshake is successful, the AT+S.SOCKON gives back the sock id: only then is it possible to write data to and read from the secure socket using AT+S.SOCKW, AT+S.SOCKQ and AT+S.SOCKR.

If the parameter ind was enabled in SOCKON, asynchronous indications of pending data from the secure socket may arrive at any time and have the format:

<CR><LF>+WIND:55:Pending Data:<sock id>:ENC<CR><LF>

When WIND:55 indications occur, the pending data is still encrypted, thus the length of decrypted data is not known in advance. The SOCKQ can be used to get the actual length of the data decrypted by TLS library and already waiting for reading.

Up to 4 consecutive WIND:55 messages (w/o SOCKR) are guaranteed. To prevent data loss, it is suggested to empty the buffer by using the AT+S.SOCKR command and to avoid exceeding 4 indications.

```
AT+S.SOCKW=<sock id>,<len>


AT+S.SOCKQ=<sock id>
AT+S.SOCKR=<sock id>,<len>
```

Note: When using a secure socket, the module can handle data packets up to 3 KB. This means that the len parameter in AT+S.SOCKW and AT+S.SOCKR can be up to 3072 bytes.

Detailed description of AT commands is contained in (see [2] in *References*).

3.4 Example 2: TLS Client with one-way authentication

This second example implements a TLS connection supporting one-way authentication. The server and CA use RSA-2048 authentication.

Before running the example, the SPWF01Sxxx (client) must be connected to the Wi-Fi LAN as illustrated in *Section 3.2* and OpenSSL must be installed on the PC (acting as server). For testing purposes, OpenSSL-1.0.1i has been used as TLS Server. Documentation and installation instructions are available on the OpenSSL website (see [12] in *References*).

The IP addresses of the client and server are automatically assigned by the network router.

To start the server, open a command prompt into the folder Project/Examples/Example2 and run this line:

```
openssl s_server -cert server_cert.pem -key server_key.pem -accept <port>
```

Note: The accept parameter specifies the TCP port to listen on for connections. If not specified, the default 4433 is used.

To start the client, it's recommended to run AT+S.TLSCERT2=clean,all to clean the Flash memory, and then use these AT commands:

```
AT+S.SETTIME=<seconds>
AT+S.TLSCERT=f_ca,<size><CR><data>
AT+S.TLSDOMAIN=f_domain,<server domain>
AT+S.SOCKON=<hostname>,<port>,s[,ind]
```

Note: Seconds is the current time expressed in seconds since 1970-Jan-01: AT+S.SETTIME can be done only once after module reset.

AT+S.TLSCERT=<f_ca/f_cert/f_key>,<size><CR><data> stores certificates or key files to the module. In case of one-way authentication, only the CA certificate (f_ca) must be loaded, the data must be in PEM format. This command accepts data after the <CR> at the end of the command line. The host is expected to supply <size> of data as last parameter of the command line. The size values must be expressed in bytes.

AT+S.TLSDOMAIN=f_domain,<server domain> passes the server domain to the module and stores the domain information to Flash memory. The server domain is a string of characters that must match the name specified in the server certificate (Common Name or others).

When using AT+S.TLSCERT and AT+S.TLSDOMAIN with f_ca, f_cert, f_key and f_domain, the file information is stored to Flash memory: since Flash memory is non-volatile, any file stored in Flash memory will remain intact even when switching off or resetting the module. Thus, to remove file information from Flash you can use the command AT+S.TLSCERT2=clean,<f_ca|f_cert|f_key|f_domain> to selectively remove a specific file, or AT+S.TLSCERT2=clean,all to remove all files information from Flash.

Note: Loading files to Flash memory is always preferred! Alternatively, you can use AT+S.TLSCERT=<ca/cert/key>,<size><CR><data> and AT+S.TLSDOMAIN=domain,<server domain> to load files to RAM, but note that:

a) RAM is volatile and in this case the loaded file information is lost after switch off or module reset

b) files stored in RAM have higher priority than those in Flash

c) file upload to RAM is deprecated in favor of Flash

AT+S.SOCKON=<hostname>,<port>,s[,ind] opens a secure socket to server hostname on port. The hostname is the IP address of the server, the port must correspond to the one specified in OpenSSL server options, and the s parameter specifies a request for secure socket. The parameter ind is optional and provides the indication (WIND:55, see below) that some data has been received: if enabled, it is strongly suggested to immediately empty the buffer when pending data is received (see SOCKQ and SOCKR commands below).

When the TLS handshake is successful, the AT+S.SOCKON gives back the sock id: only then is it possible to write data to and read from the secure socket using AT+S.SOCKW, AT+S.SOCKQ and AT+S.SOCKR.

If the parameter ind was enabled in SOCKON, asynchronous indications of pending data from the secure socket may arrive at any time and have the format:

```
<CR><LF>+WIND:55:Pending Data:<sock id>:ENC<CR><LF>
```

When WIND:55 indications occur, the pending data is still encrypted, thus the length of decrypted data is not known in advance. The SOCKQ can be used to get the actual length of the data decrypted by TLS library and already waiting for reading.

Up to 4 consecutive WIND:55 messages (w/o SOCKR) are guaranteed. To prevent data loss, it is suggested to empty the buffer by using the AT+S.SOCKR command and to avoid exceeding 4 indications.

```
AT+S.SOCKW=<sock id>,<len>


AT+S.SOCKQ=<sock id>
AT+S.SOCKR=<sock id>,<len>
```

Note: when using a secure socket, the module can handle data packets up to 3 KB. This means that the len parameter in AT+S.SOCKW and AT+S.SOCKR can be up to 3072 bytes.

Detailed description of AT commands is contained in (see [2] in *References*).

3.5 Example 3: Gmail SMTP server access with anonymous negotiation

To protect SMTP communications, server can use SSL/TLS encryption to provide authentication and information encryption. To enable the SMTP client to verify the SMTP server certificate, the issuing CA certificates should be made available to the client. In this case the CA certificate is not provided and the client is not expected to verify server certificate. Even the SMTP client is not providing its certificate, so the negotiation is anonymous.

For the purpose of the example, the Gmail SMTP server is used. Since it requires SMTP authorization, a Gmail account is needed for <username> (the full Gmail address, e.g. example@gmail.com) and <password> options.

To access SMTP server, it is recommended to run AT+S.TLSCERT2=clean,all to clean the Flash memory and then open a secure connection on port 465:

```
AT+S.SOCKON=smtp.gmail.com,465,s
```

In this example, the parameter ind is omitted: if enabled, this option requires to read the socket when a pending indication message is received.

When the TLS handshake is successful, the AT+S.SOCKON gives back the sock id: only then is it possible to use the AT+S.SOCKW to make the login on Gmail server by sending first “EHLO” command and then “AUTH PLAIN” with your credentials using the base64 encoded username/password.

3.6 Example 4: Xively example with anonymous negotiation

Xively is a cloud platform for developing and managing commercial services on the Internet of Things. Xively ensures protected communication channels by using SSL/TLS authentication and encryption. In this example the CA certificate is not provided and the client is not expected to verify the server certificate. The client is not providing its certificate, so the negotiation is anonymous.

To access Xively server, it is recommended to run AT+S.TLSCERT2=clean,all to clean the Flash memory and then open a secure connection on port 8091:

```
AT+S.SOCKON=api.xively.com,8091,s
```

In this example the parameter ind is omitted: if enabled, this option requires to read the socket when a pending indication message is received.

3.7 Example 5: connect to HTTPS my.st.com

This example shows how to use the Wi-Fi module as a TLS client to connect to a secure HTTPS server and to send/receive HTTP requests/replies.

The HTTPS server we want to connect to is https://my.st.com.

In this example we are not verifying the server’s certificate (anonymous negotiation), so we are not uploading any certificate to the module. In case you need to authenticate the server, you have to configure the module as in example 2 (properly set the appropriate CA certificate, domain name and time).

The first step is to open the secure connection to the HTTPS server: it is recommended to run AT+S.TLSCERT2=clean,all to clean the Flash memory and then open a secure connection to hostname my.st.com on port 443 as below. Note that the parameter ind is enabled to activate asynchronous indications of pending data from the server.

```
AT+S.SOCKON=my.st.com,443,s,ind<CR>
<CR><LF>
 ID: 00<CR><LF>
<CR><LF>
OK
```

When the TLS Handshake is successful, the AT+S.SOCKON gives back the sock id and all the data exchanged from now on are encrypted with the newly established session key.

The second step is to send an HTTPS message to the server: the following code shows how to use the AT+S.SOCKW to send a basic HTTP GET request:

```
AT+S.SOCKW=00,18<CR>
GET / HTTP/1.1<CR><LF>
<CR><LF>
OK
```

The server response to the above GET request is signaled by a couple of asynchronous indications (see the WIND:55 in the box below). In order to get and decrypt the received data, we have to iteratively call AT+S.SOCKQ and AT+S.SOCKR to process all the received pending data as follows:

```
+WIND:55:Pending Data:0:ENC<CR><LF>
<CR><LF>
+WIND:55:Pending Data:0:ENC<CR><LF>


AT+S.SOCKQ=00<CR>
<CR><LF>
 DATALEN: 483<CR><LF>
<CR><LF>
OK


AT+S.SOCKR=00,483<CR>

HTTP/1.1 302 Found<CR><LF>
Date: Tue, 31 Mar 2015 08:48:16 GMT<CR><LF>
Location: https://my.st.com/cas/login?service=https://my.st.com/<CR><LF>
Content-Length: 238<CR><LF>
Content-Type: text/html; charset=iso-8859-1<CR><LF>
Proxy-Connection: Keep-Alive<CR><LF>
Connection: Keep-Alive<CR><LF>
<CR><LF>
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"><LF>
<html><head><LF>
```

```
<title>302 Found</title><LF>
</head><body><LF>
<h1>Found</h1><LF>
<p>The document has moved <a
href="https://my.st.com/cas/login?service=https://my.st.com/">here</a>.</p
><LF>
</body></html><LF><CR>
<LF>
OK


AT+S.SOCKQ=00<CR>
<CR><LF>
 DATALEN: 0<CR><LF>
<CR><LF>
OK
```

4 Known limitations and revisions

Due to PODDLE vulnerability (see [13] in *References*), there are plans to drop SSL3.0 support in future releases of this firmware.

5 Glossary

AES	Advanced encryption standard
Camelia	Block cipher developed by Mitsubishi and NTT
DES	Data encryption standard
DH	Diffie-Hellman
DHE	Diffie-Hellman ephemeral
DSA	Digital signature algorithm
DSS	Digital signature standard
ECDH	Elliptic-curve Diffie-Hellman
ECDSA	Elliptic curve digital signature algorithm
HMAC	keyed-hash message authentication code
HTTPS	Hypertext transfer protocol secure
IANA	Internet assigned numbers authority
IDEA	International data encryption algorithm
IETF	Internet engineering task force
KRB5	Kerberos
MAC	Message authentication code
MD5	Message digest algorithm 5
PSK	Pre-shared key
RSA	Rivest, Shamir, Adleman
RC4	Rivest cipher 4
SHA	Secure hash algorithm
SRP	Secure remote password protocol
SSL	Secure sockets layer
TLS	Transport layer security
3DES	Triple data encryption algorithm

6 References

1. SPWF01Sxxx WiFi module, www.st.com/wifimodules
2. UM1695 - “Command set reference guide for AT full stack for SPWF01Sx series of Wi-Fi modules”, User Manual of SPWF01Sxxx,
3. The “Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, http://tools.ietf.org/html/rfc5246
4. “Transport Layer Security”, Wikipedia, http://en.wikipedia.org/wiki/Transport_Layer_Security
5. http://www.symantec.com/connect/articles/apache-2-ssltls-step-step-part-1
6. IANA registry, http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml
7. “Certificates and Authentication”, RedHat portal, https://access.redhat.com/site/documentation/en-US/Red_Hat_Certificate_System/8.0/html/Deployment_Guide/Introduction_to_Public_Key_Cryptography-Certificates_and_Authentication.html
8. “A String Representation of Distinguished Names”, RFC 4514, http://www.ietf.org/rfc/rfc4514.txt.
9. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, RFC 3280, http://tools.ietf.org/html/rfc3280
10. “Digital certificates”, IBM information center, http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=%2Fcom.ibm.mq.csqzas.doc%2Fsy10600_.htm
11. Server Name Indication, Wikipedia, http://en.wikipedia.org/wiki/Server_Name_Indication
12. OpenSSL website, www.openssl.org
13. CVE-2014-3566, CVE list of security vulnerabilities, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566, 14/Oct-2014.

Appendix A Certificate generation with OpenSSL

OpenSSL is an open-source implementation for PC platforms (Win, *nix, Mac) of the SSL/TLS protocols providing both client and server functionalities. The core library, written in the C programming language, implements the basic cryptographic functions and provides various utility functions. For testing purposes, OpenSSL-1.0.1i has been used as TLS Server. For documentation and installation instructions, please refer to OpenSSL website (see [12] in *References*).

Just for example purposes, included below are the OpenSSL commands to generate RSA and EC compatible certificates and associated private keys.

Example for generating RSA signed certificates:

Generate key pair and a self-signed certificate for the CA (trusted certificate):

1. openssl genpkey -out ca_key.pem -outform PEM -algorithm rsa -pkeyopt rsa_keygen_bits:1024
2. openssl req -new -key ca_key.pem -days 6500 -set_serial 1111 -subj “/C=IT/ST=Lombardia/L=Milan/O=STM/OU=R&D/CN=CA domain” -out ca_cert.pem -x509

Generate server certificate/key pair:

3. openssl genpkey -out server_key.pem -outform PEM -algorithm rsa -pkeyopt rsa_keygen_bits:1024
4. openssl req -new -key server_key.pem -days 6500 -set_serial 2222 -subj “/C=IT/ST=Lombardia/L=Milan/O=STM/OU=R&D/CN=server domain” -out server_cert_req.pem
5. openssl ca -in server_cert_req.pem -out server_cert.pem -days 6500 -keyfile ca_key.pem -cert ca_cert.pem -notext -batch

Generate client certificate/key pair:

6. openssl genpkey -out client_key.pem -outform PEM -algorithm rsa -pkeyopt rsa_keygen_bits:1024
7. openssl req -new -key client_key.pem -days 6500 -set_serial 3333 -subj “/C=IT/ST=Lombardia/L=Milan/O=STM/OU=R&D/CN=client domain” -out client_cert_req.pem
8. openssl ca -in client_cert_req.pem -out client_cert.pem -days 6500 -keyfile ca_key.pem -cert ca_cert.pem -notext -batch

Example for generating ECC signed certificates:

Generate key pair and a self-signed certificate for the CA (trusted certificate):

1. openssl ecparam -out ca_key.pem -name prime192v1 -genkey
2. openssl req -new -key ca_key.pem -days 6500 -set_serial 1111 -subj “/C=IT/ST=Lombardia/L=Milan/O=STM/OU=R&D/CN=CA domain” -out ca_cert.pem -x509

7 Revision history

Table 4. Document revision history

Date	Revision	Changes
07-May-2015	1	Initial release.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.