



ESP8266 SSL User Manual

Version 1.4

Espressif Systems IOT Team

<http://bbs.espressif.com/>

Copyright © 2016



Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice.

THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

The WiFi Alliance Member Logo is a trademark of the WiFi Alliance.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2016 Espressif Systems (Shanghai) Pte. Ltd. All rights reserved.



Table of Contents

- 1. Preambles.....4
- 2. ESP8266 as SSL server.....5
 - 2.1. Generate certificate.....5
- 3. ESP8266 as SSL client.....9
 - 3.1. Generate CA Certificate9
- 4. Software APIs.....10
 - 4.1. espconn_secure_ca_enable.....10
 - 4.2. espconn_secure_ca_disable.....11
 - 4.3. espconn_secure_cert_req_enable12
 - 4.4. espconn_secure_cert_req_disable12
 - 4.5. espconn_secure_set_default_certificate.....13
 - 4.6. espconn_secure_set_default_private_key13
 - 4.7. espconn_secure_accept14
 - 4.8. espconn_secure_delete15
 - 4.9. espconn_secure_set_size15
 - 4.10. espconn_secure_get_size.....16
 - 4.11. espconn_secure_connect17
 - 4.12. espconn_secure_send17
 - 4.13. espconn_secure_disconnect18



1.

Preambles

This manual introduces how to implement SSL encryption based on ESP8266_NONOS_SDK when ESP8266 runs as either SSL server or SSL client.

SSL function usually requires a lot of RAM memory, therefore, users need to make sure that there is enough space before running the application. If the space occupied by SSL buffer is 8KB defined by [espconn_secure_set_size](#), then at least 22KB memory size is required if SSL function is to be implemented. The specific memory size required varies with the actual size of data sending from SSL server.

If the SSL bi-directional verification is enabled, the SSL buffer size allowed to set by [espconn_secure_set_size](#) is 3072 bytes at most. The specific size allowed to set depends on the actual size of heap available.

Demo and scripts on how to generate SSL certificate: `test_cert_and_creat_information` and `TLS_BiDirectVerif_Demo`.

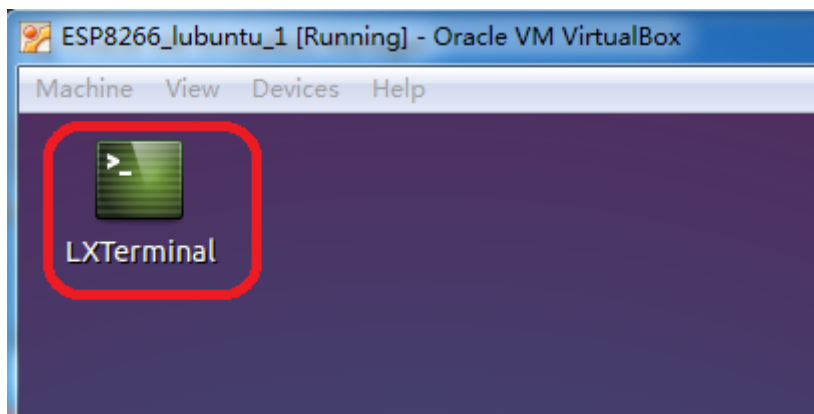
2. ESP8266 as SSL server

When ESP8266 is running as a SSL server, header files `cert.h` and `private_key.h` required for SSL encryption can be generated when encryption certificate is provided. Users can refer to sample codes defined by macro definition `#define SERVER_SSL_ENABLE` in `IoT_Demo` on how to implement SSL server.

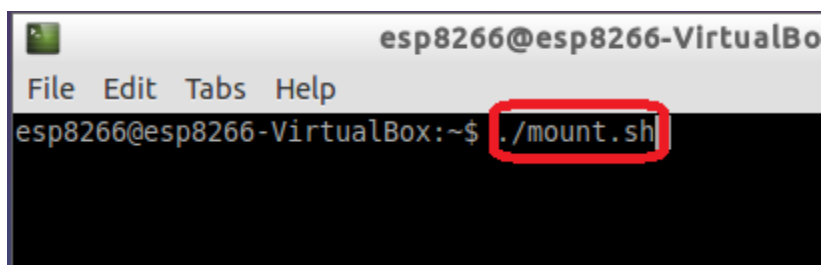
CA verification function is disabled by default, however, it can be enabled by calling `espconn_secure_ca_enable`.

2.1. Generate certificate

- (1) Copy script "`makefile.sh`" to Lubuntu virtual box share folder.
 - Please refer to BBS <http://bbs.espressif.com/viewtopic.php?f=21&t=86> on how to set up linux (Lubuntu) compile environment.
- (2) Mount the share folder
 - Open "LXTerminal" in virtual box



- Enter command `./mount.sh`, press "Enter" key.

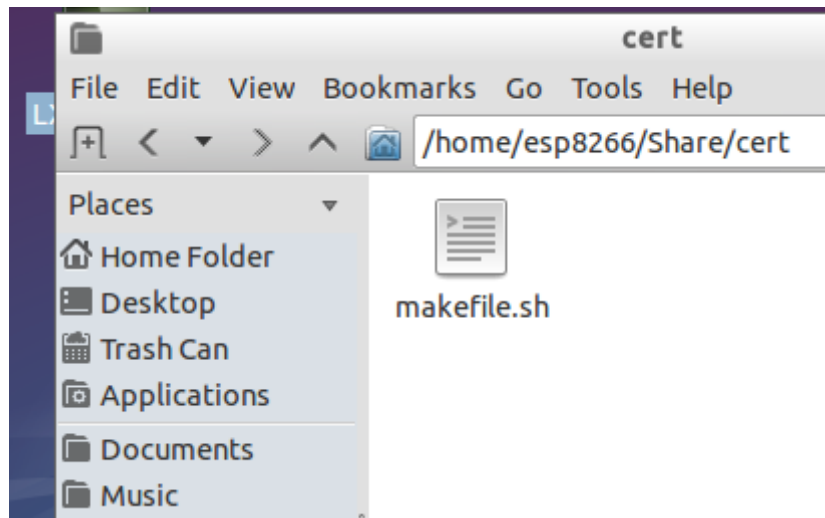


- Enter password: **espressif**, press "Enter" key.



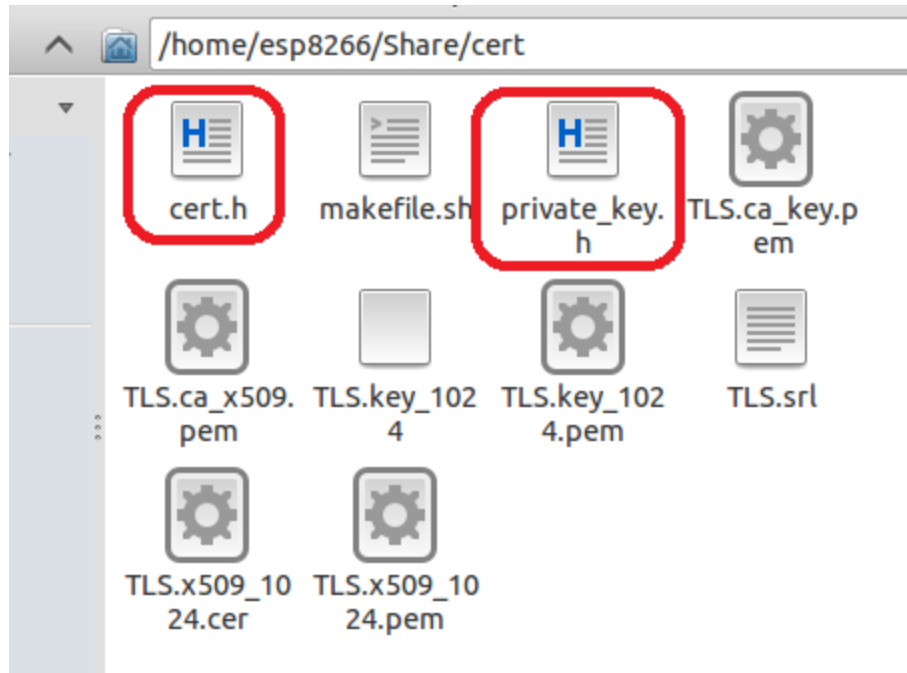
```
esp8266@esp8266-VirtualBox: ~
File Edit Tabs Help
esp8266@esp8266-VirtualBox:~$ ./mount.sh
[sudo] password for esp8266:
esp8266@esp8266-VirtualBox:~$
```

(3) Open share folder in virtual box, and get script "makefile.sh" there.



(4) Enter command `./makefile.sh`, and run script "makefile.sh", then two header files `cert.h` and `private_key.h` will be generated. Please use these two header files according to the IoT Demo.

```
esp8266@esp8266-VirtualBox:~$ cd /home/esp8266/Share/cert
esp8266@esp8266-VirtualBox:~/Share/cert$ ./makefile.sh
```



Note:

- IP address in script "makefile.sh" should be the actual SSL server IP address, as is shown in the picture below:

```
cat > certs.conf << EOF
[ req ]
distinguished_name = req_distinguished_name
prompt = no

[ req_distinguished_name ]
0 = $PROJECT_NAME
CN = 127.0.0.1
EOF
```



- Script "makefile.sh" adopts 1024 bit encryption algorithm by default. If users need to adopt 512 bit encryption algorithm, please change the number 1024 in script "makefile.sh" to 512.

```
# private key generation
openssl genrsa -out TLS.ca_key.pem 1024
openssl genrsa -out TLS.key_1024.pem 1024

# convert private keys into DER format
openssl rsa -in TLS.key_1024.pem -out TLS.key_1024 -outform DER

# cert requests
openssl req -out TLS.ca_x509.req -key TLS.ca_key.pem -new \
    -config ./ca_cert.conf
openssl req -out TLS.x509_1024.req -key TLS.key_1024.pem -new \
    -config ./certs.conf

# generate the actual certs.
openssl x509 -req -in TLS.ca_x509.req -out TLS.ca_x509.pem \
    -sha1 -days 5000 -signkey TLS.ca_key.pem
openssl x509 -req -in TLS.x509_1024.req -out TLS.x509_1024.pem \
    -sha1 -CAcreateserial -days 5000 \
    -CA TLS.ca_x509.pem -CAkey TLS.ca_key.pem

# some cleanup
rm TLS*.req
rm *.conf

openssl x509 -in TLS.ca_x509.pem -outform DER -out TLS.ca_x509.cer
openssl x509 -in TLS.x509_1024.pem -outform DER -out TLS.x509_1024.cer

#
# Generate the certificates and keys for encrypt.
#
```

Notice:

- Since [ESP8266_NONOS_SDK_V1.4.0](#), users should call [espconn_secure_set_default_certificate](#) and [espconn_secure_set_default_private_key](#) to set SSL certificate and secure key.
- SSL server certificate generated above is issued by Espressif Systems, not CA. Users who requires CA certificate can add [TLS.ca_x509.cer](#) which generated as above into SSL client's trust anchor, then generate [esp_ca_cert.bin](#) by script "**make_cacert.py**" according to **3.1 Generate CA Certificate**, and download [esp_ca_cert.bin](#) generated by CA certificate into the corresponding addresses in the flash.



3. ESP8266 as SSL client

Sample code of ESP8266 running as SSL client is defined by macro definition `#define CLIENT_SSL_ENABLE` in IOT_Demo. When running as a client, bi-directional verification is supported. CA verification function is disabled by default, however, it can be invoked by calling function `espconn_secure_ca_enable`.

Users can refer to demonstration of SSL certificate, TLS_BiDirectVerif_Demo.

3.1. Generate CA Certificate

- (1) Revise script "`makefile.sh`" and generate a CA certificate issued by developers themselves. For example, `TLS.ca_x509.cer`, as is shown in the "TLS_BiDirectVerif_Demo".
- (2) Generate a SSL client certificate using the CA certificate issued. For example, as is shown in the "TLS_BiDirectVerif_Demo", the `TLS.x509_1024.cer`.
- (3) Take out the secure key that is used during SSL certificate generation. For example, `TLS.key_1024`, as is shown in the "TLS_BiDirectVerif_Demo".
- (4) Move script "`make_cacert.py`" and CA certificate files (for example, `TLS.ca_x509.cer`) to the same directory.
- (5) Run script "`make_cacert.py`", then it will combine with CA files in the same directory and generate `esp_ca_cert.bin`. The address that `esp_ca_cert.bin` will be written need to be set by calling `espconn_secure_ca_enable`.
- (6) Rename the certificate (such as `TLS.x509_1024.cer`) as `certificate.cer`; rename the secure key (such as `TLS.key_1024`) as `private_key.key_1024`. Please be noted that both the certificates and the secure keys should be renamed in this procedure, otherwise certification will fail.
- (7) Copy and move the renamed files to the same directory of "`make_cert.py`".
- (8) Run script "`make_cert.py`" and `esp_cert_private_key.bin` will be generated. The address that `esp_cert_private_key.bin` will be written is set by calling `espconn_secure_cert_req_enable`.



4.

Software APIs

SSL related APIs are different from normal TCP APIs and must not be used interchangeably. In SSL connection, only the below APIs can be used:

- `espconn_secure_XXX` APIs which are SSL related APIs;
- `espconn_regist_XXX` APIs to register callbacks, except `espconn_regist_write_finish`;
- `espconn_port` to get an available port.

Herein this manual, only `espconn_secure_XXX` APIs are introduced in detail. For more information about other software APIs please refer to documentation "2C-ESP8266__SDK__Programming Guide".

One demo of SSL connection can be found via: <http://bbs.espressif.com/viewtopic.php?f=21&t=389>

4.1. `espconn_secure_ca_enable`

Function:

Enable SSL CA (certificate authenticate) function

Note:

- CA function is disabled by default
- `esp_ca_cert.bin` must be written when this API is called
- If user want to call this API, please call the below APIs before encryption (SSL) is established:
 - ESP8266 as TCP SSL server: call this API before `espconn_secure_accept` is called;
 - ESP8266 as TCP SSL client: call this API before `espconn_secure_connect` is called.

Prototype:

```
bool espconn_secure_ca_enable (uint8 level, uint32 flash_sector)
```

Parameter:

`uint8 level` : set configuration for ESP8266 SSL server/client:

`0x01` SSL client;



0x02 SSL server;

0x03 both SSL client and SSL server

uint32 flash_sector : Flash sector in which CA certificate (esp_ca_cert.bin) is written into. For example, parameters 0x3B should be written into Flash 0x3B000 in the flash. Please be noted that sectors used for storing codes and system parameters must not be covered.

Return:

true : succeed

false : fail

4.2. espconn_secure_ca_disable

Function:

Disable SSL CA verification function

Note:

- CA verification function is disabled by default.

Prototype:

bool espconn_secure_ca_disable (uint8 level)

Parameter:

uint8 level : when ESP8266 runs as SSL server/client:

0x01 SSL client;

0x02 SSL server;

0x03 SSL client and SSL server

Return:

true : succeed

false : fail



4.3. `espconn_secure_cert_req_enable`

Function:

Enable certification verification function when ESP8266 runs as SSL client.

Note:

- Certification verification function is disabled by defaults. If the SSL server need not certification verification, this API does not need to be called.
- Call this API before `espconn_secure_connect` is called.

Prototype:

`bool espconn_secure_cert_req_enable (uint8 level, uint32 flash_sector)`

Parameter:

`uint8 level` : can only be set as `0x01` when ESP8266 runs as SSL client;

`uint32 flash_sector` : set the address where secure key (`esp_cert_private_key.bin`) will be written into the flash. For example, parameters `0x3A` should be written into Flash `0x3A000` in the flash. Please be noted that sectors used for storing codes and system parameters must not be covered.

Return:

true : succeed

false : fail

4.4. `espconn_secure_cert_req_disable`

Function:

Disable certification verification function when ESP8266 runs as SSL client

Note:

- Certification verification function is disabled by default

Prototype:

`bool espconn_secure_ca_disable (uint8 level)`



Parameter:

`uint8 level` : can only be set as `0x01` when ESP8266 runs as SSL client;

Return:

true : succeed

false : fail

4.5. `espconn_secure_set_default_certificate`

Function:

Set the certificate when ESP8266 runs as SSL server

Note:

- Demos can be found in `ESP8266_NONOS_SDK\examples\IoT_Demo`
- This API has to be called before `espconn_secure_accept`.

Prototype:

```
bool espconn_secure_set_default_certificate(const uint8_t* certificate, uint16_t length)
```

Parameter:

`const uint8_t* certificate` : pointer of the certificate

`uint16_t length` : length of the certificate

Return:

true : succeed

false : fail

4.6. `espconn_secure_set_default_private_key`

Function:

Set the secure key when ESP8266 runs as SSL server

Note:

- Demos can be found in `ESP8266_NONOS_SDK\examples\IoT_Demo`.
- This API has to be called before `espconn_secure_accept`.



Prototype:

`bool espconn_secure_set_default_private_key (const uint8_t* key, uint16_t length)`

Parameter:

`const uint8_t* key` : pointer of the secure key

`uint16_t length` : length of the secure key

Return:

true : succeed

false : fail

4.7. `espconn_secure_accept`

Function:

Create an SSL TCP server, intercept SSL handshake.

Note:

- This API can be called only once, only one SSL server is allowed to be created, and only one SSL client can be connected.
- If the size of SSL encrypted data patch is larger than the buffer size defined by `espconn_secure_set_size`, and is beyond the processing capability of ESP8266, then SSL will be disconnected, and callback function `espconn_reconnect_callback` will be invoked.
- Users should call API `espconn_secure_set_default_certificate` and `espconn_secure_set_default_private_key` to set SSL certificate and secure key first.

Prototype:

`sint8 espconn_secure_accept(struct espconn *espconn)`

Parameter:

`struct espconn *espconn` : architecture of the web connection

Return:

0 : succeed

Non-0 : fail, errors will be returned



`ESPCONN_ARG` - TCP connection corresponding with parameter `espconn` cannot be found

`ESPCONN_MEM` - memory size is not enough

`ESPCONN_ISCONN` - connection succeeded

4.8. `espconn_secure_delete`

Function:

Delete the SSL connection when ESP8266 runs as SSL server.

Prototype:

```
sint8 espconn_secure_delete(struct espconn *espconn)
```

Parameter:

`struct espconn *espconn` : corresponding SSL connection

Return:

0 : succeed

Non-0 : error, return error code

`ESPCONN_ARG` - illegal argument, can't find network transmission according to structure `espconn`

`ESPCONN_INPROGRESS` - the SSL connection is still in progress, please call `espconn_secure_disconnect` to disconnect before deleting it.

4.9. `espconn_secure_set_size`

Function:

Set buffer size of encrypted data (SSL)

Note:

Buffer size default to be 2KBytes. Before modification, please call this API before encryption (SSL) connection is established:

- ▶ ESP8266 as TCP SSL server: call this API before `espconn_secure_accept` is called;



- ESP8266 as TCP SSL client: call this API before `espconn_secure_connect` is called.

Prototype:

`bool espconn_secure_set_size (uint8 level, uint16 size)`

Parameters:

`uint8 level` : set buffer for ESP8266 SSL server/client:

- `0x01` SSL client;
- `0x02` SSL server;
- `0x03` both SSL client and SSL server

`uint16 size` : buffer size, range:1 ~ 8192, unit:byte, default is 2048

Return:

`true` : succeed
`false` : fail

4.10. `espconn_secure_get_size`

Function:

Get buffer size of encrypted data (SSL)

Prototype:

`sint16 espconn_secure_get_size (uint8 level)`

Parameters:

`uint8 level` : buffer for ESP8266 SSL server/client:

- `0x01` SSL client;
- `0x02` SSL server;
- `0x03` both SSL client and SSL server

Return:

buffer size



4.11. `espconn_secure_connect`

Function:

Secure connect (SSL) to a TCP server (ESP8266 is acting as TCP client.)

Note:

- Only one connection is allowed when ESP8266 as SSL client, please call `espconn_secure_disconnect` first, if you want to create another SSL connection.
- If SSL encrypted packet size is larger than ESP8266 SSL buffer size (default 2KB, set by `espconn_secure_set_size`), SSL connection will fail, will enter `espconn_reconnect_callback`

Prototype:

`sint8 espconn_secure_connect (struct espconn *espconn)`

Parameters:

`struct espconn *espconn` : corresponding connected control block structure

Return:

0 : succeed

Non-0 : error code

`ESPCONN_MEM` - Out of memory

`ESPCONN_ISCONN` - Already connected

`ESPCONN_ARG` - illegal argument, can't find TCP connection according to structure `espconn`

4.12. `espconn_secure_send`

Function: send encrypted data (SSL)

Note:

Please call `espconn_secure_send` after `espconn_sent_callback` of the pre-packet.



Prototype:

```
sint8 espconn_secure_send (  
    struct espconn *espconn,  
    uint8 *psent,  
    uint16 length  
)
```

Parameters:

[struct espconn *espconn](#) : corresponding connected control block structure

[uint8 *psent](#) : sent data pointer

[uint16 length](#) : sent data length

Return:

0 : succeed

Non-0 : error code [ESPCONN_ARG](#) - illegal argument, can't find TCP connection according to structure [espconn](#)

4.13. [espconn_secure_disconnect](#)

Function: secure TCP disconnection (SSL)

Prototype:

```
sint8 espconn_secure_disconnect(struct espconn *espconn)
```

Parameters:

[struct espconn *espconn](#) : corresponding connected control block structure

Return:

0 : succeed

Non-0 : error code [ESPCONN_ARG](#) - illegal argument, can't find TCP connection according to structure [espconn](#)