AN0150
Application Note

AT32 HTTPS Server Based on Mbed TLS

Introduction

Traditionally, HTTP is designed as a clear text protocol, which means that the data sent may be intercepted by a third party through packet sniffing attack, threatening the security of data transmission and even causing great losses if sensitive data such as bank account password is revealed. To guarantee the security of data transmission, HTTPS (Hypertext Transfer Protocol Secure), a secure version of the HTTP protocol, is introduced to authenticate the identities of both transmission parties and encrypt the transmitted content, so as to prevent data sent over the Internet from being intercepted and read by a third party.
This application note introduces how to use mbed TLS to build a HTTPS server to allow users to write web content as needed and encrypt the transmitted data at the same time.

Applicable products:

Part number	AT32F407xx
	AT32F437xx

Contents

List of tables

List of figures

1 HTTPS introduction

The security of HTTPS is heavily dependent on the Transport Layer Security (TLS). TLS is a cryptographic protocol that encrypts the communication over Internet. As a handshake protocol of Secure Sockets Layer (SSL), TLS provides authentication between the client and server. TLS certificates require a public key, server identity, and signature of the certificate issuer. The corresponding private key is never to be revealed. Any key data encrypted with the private key can only be decrypted with the public key, and vice versa. The encrypted communication process is shown in Figure 1.

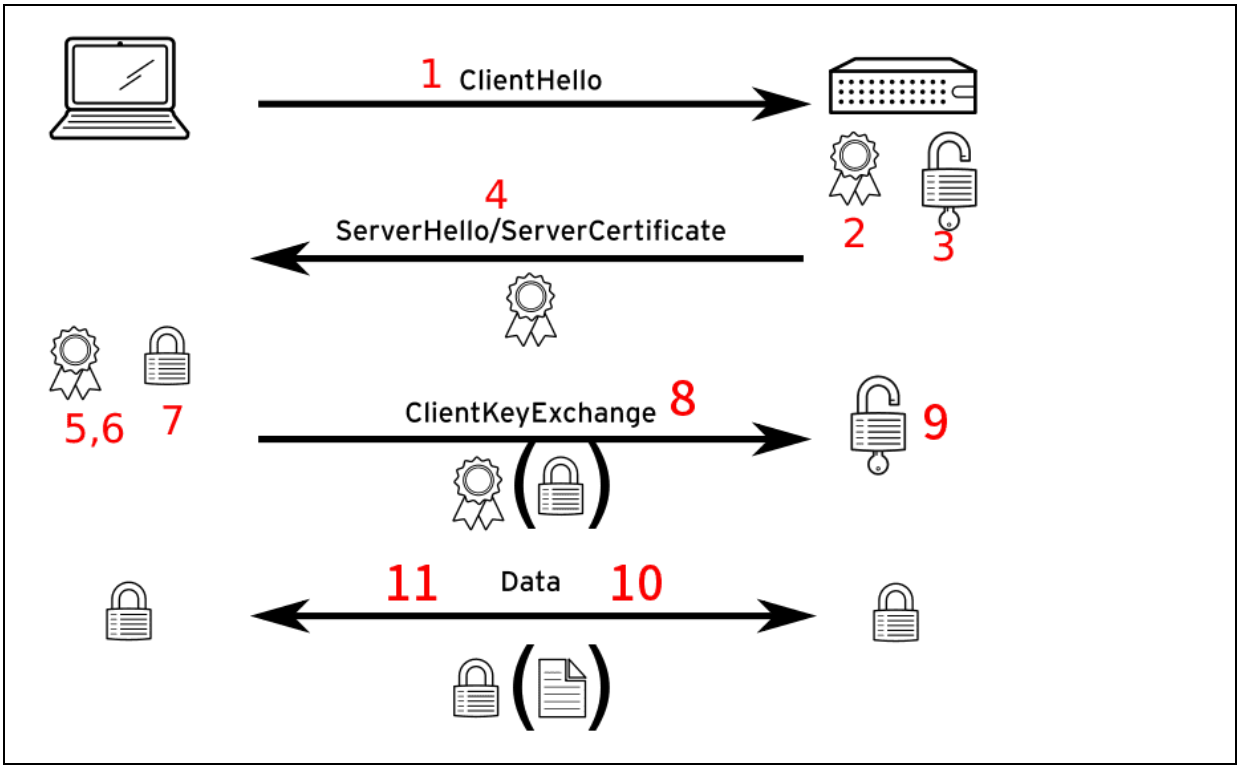
1. The client sends a “client hello” message to the server, including
   - A 32-bit random digital “client_random” with timestamp
   - Encryption protocol
   - Encryption modes supported by the client
2. The server requires a certificate that is issued by a trusted organization. The certificate also can be the personal certificate which should be authenticated by the client.
3. A pair of public key and private key. The client uses the public key to lock important data and then send to the server. The locked data can only be unlocked with the private key of the server, and cannot be cracked even if it is intercepted during transmission.
4. The server responds to the “client hello” with the below message:
   - A 32-bit random digital “server_random” with timestamp
   - Encryption protocol
   - Encryption mode
   - Server certificate, which contains the owner name, website address, certificate public key, certificate authority digital signature, expiration time, etc.
5. The client verifies the server’s digital certificate, including the certificate authority and expiration. A warning message pops up in case of any exception (in this application note, we use the self-signed certificate, not the certificate issued by a third party certificate authority (CA); therefore, the client should get the public key (“kvm5.pem” in the next section) of the issuing authority to verify the authority signature).
6. All TLS handshake messages sent before are in clear text. The server’s certificate is received and certified, and the client generates “PreMaster_Secret”.
   - Use encryption algorithm, such as RSA and Diffie-Hellman, for server_random operation.
   - Or generate “PreMaster_Key”.
   - A 48-bit Key, where the first 2 bytes are the protocol version number and the last 46 bytes are the random number used in the symmetric encryption key.
7. The client uses the public key from the server to encrypt “PreMaster_Secret”.
8. The client sends the encrypted “PreMaster_Secret” to the server to allow the server to generate “Master_Secret” with random values.
9. The server decrypt “PreMaster_Secret” with the private key.
   - Both client and server have the same “PreMaster_Secret” and random values (client_random, server_random).
   - Use “client_random” and “server_random” as seeds and combine with

"PreMaster_Secret", and the client and server calculate the same "Master_Secret".

- Key Material for data encryption and decryption.

10. Key Material for data encryption and decryption.
11. Key Material for data encryption and decryption.

Figure 1. HTTPS encrypted communication principle

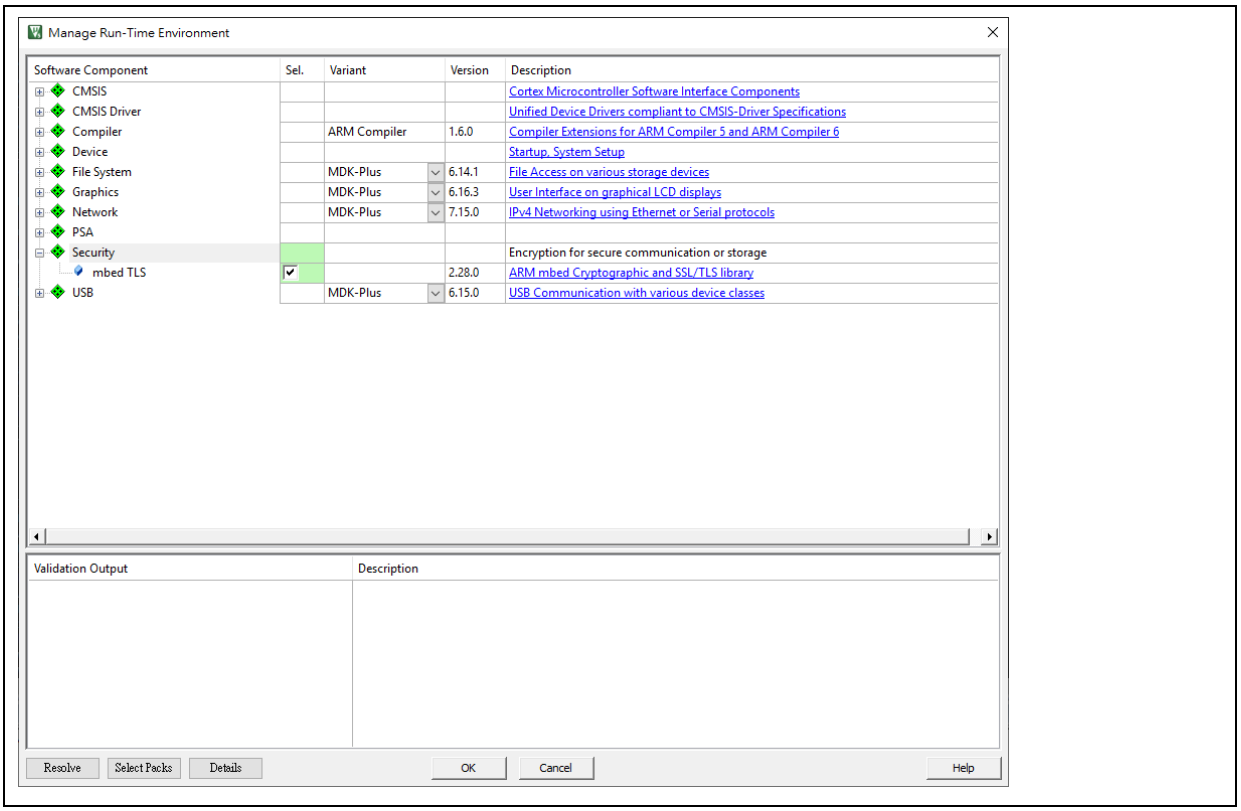


2 Example: HTTPS server

2.1 Function overview

In this example, the EMAC feature is used together with LwIP protocol stack. The LwIP protocol stack provides HTTPS API, and the encryption function of mbed TLS is also used. This package is available from Keil's official website or through Keil's Pack Installer.

Figure 2. Add mbed TLS in Manage Run-Time environment



2.2 Resources

1) Hardware

   AT-START BOARD of the corresponding series

2) Software

   at32f4xx\project\mdk_v5

2.3 Software design

1) Configuration
   - Configure EMAC feature
   - Initialize LwIP protocol stack
   - Configure private key and CA certificate to the server
   - Initialize HTTPS server
2) Code
   - ssl_server function code

```
void ssl_server(void const *argument)
{
    int ret, len;



#ifdef MBEDTLS_MEMORY_BUFFER_ALLOC_C
    mbedtls_memory_buffer_alloc_init(memory_buf, sizeof(memory_buf));
#endif
    //mbedtls_net_init( &listen_fd );
    listen_fd.fd = -1;
    //mbedtls_net_init( &client_fd );
    client_fd.fd = -1;
    mbedtls_ssl_init( &ssl );
    mbedtls_ssl_config_init( &conf );
#if defined(MBEDTLS_SSL_CACHE_C)
    mbedtls_ssl_cache_init( &cache );
#endif
    mbedtls_x509_crt_init( &srvcert );
    mbedtls_pk_init( &pkey );
    mbedtls_entropy_init( &entropy );
    mbedtls_ctr_drbg_init( &ctr_drbg );

#if defined(MBEDTLS_DEBUG_C)
    mbedtls_debug_set_threshold( DEBUG_LEVEL );
#endif


    /*
     * 1. Load the certificates and private RSA key
     */
    mbedtls_printf( "\r\n    . Loading the server cert. and key..." );

    /*
     * This demonstration program uses embedded test certificates.
     * Using mbedtls_x509_crt_parse_file() to read the server and CA certificates
     * requires the implmentation of the File I/O API using the FatFs, that is
     * not implemented yet.
     */
```

```
  ret = mbedtls_x509_crt_parse( &srvcert, (const unsigned char *) mbedtls_test_srv_crt,
mbedtls_test_srv_crt_len );
  if(ret != 0)
  {
    mbedtls_printf( " failed\r\n  !  mbedtls_x509_crt_parse returned %d\r\n", ret );
    goto exit;
  }

  ret =  mbedtls_pk_parse_key(&pkey, (const unsigned char *) mbedtls_test_srv_key,
mbedtls_test_srv_key_len, NULL, 0);

  if( ret != 0 )
  {
    mbedtls_printf(" failed\r\n  !  mbedtls_pk_parse_key returned %d\r\n", ret);
    goto exit;
  }

  mbedtls_printf( " ok\r\n" );

  /*
   * 2. Setup the listening TCP socket
   */
  mbedtls_printf( "  . Bind on https://localhost:443/ ..." );

  if((ret = mbedtls_net_bind(&listen_fd, NULL, "443", MBEDTLS_NET_PROTO_TCP )) != 0)
  {
    mbedtls_printf( " failed\r\n  ! mbedtls_net_bind returned -0x%x\r\n", -ret );
    goto exit;
  }

  mbedtls_printf( " ok\r\n" );

  /*
   * 3. Seed the RNG
   */
  mbedtls_printf( "  . Seeding the random number generator..." );

  if((ret = mbedtls_ctr_drbg_seed(&ctr_drbg, ALTCP_MBEDTLS_RNG_FN, &entropy, (const
unsigned char *) pers, strlen( (char *)pers))) != 0)
```

```
  {
    mbedtls_printf( " failed\r\n  ! mbedtls_ctr_drbg_seed returned -0x%x\r\n", -ret );
    goto exit;
  }

  mbedtls_printf( " ok\r\n" );

  /*
   * 4. Setup stuff
   */
  mbedtls_printf( "  . Setting up the SSL data...." );

  if((ret = mbedtls_ssl_config_defaults(&conf, MBEDTLS_SSL_IS_SERVER,
MBEDTLS_SSL_TRANSPORT_STREAM, MBEDTLS_SSL_PRESET_DEFAULT)) != 0)
  {
    mbedtls_printf( " failed\r\n  ! mbedtls_ssl_config_defaults returned -0x%x\r\n", -ret );
    goto exit;
  }

  mbedtls_ssl_conf_rng(&conf, mbedtls_ctr_drbg_random, &ctr_drbg);

#if defined(MBEDTLS_SSL_CACHE_C)
  mbedtls_ssl_conf_session_cache(&conf, &cache, mbedtls_ssl_cache_get,
mbedtls_ssl_cache_set);
#endif

  mbedtls_ssl_conf_ca_chain(&conf, srvcert.next, NULL);
  if((ret = mbedtls_ssl_conf_own_cert(&conf, &srvcert, &pkey)) != 0)
  {
    mbedtls_printf( " failed\r\n  ! mbedtls_ssl_conf_own_cert returned -0x%x\r\n", -ret );
    goto exit;
  }

  if((ret = mbedtls_ssl_setup(&ssl, &conf)) != 0)
  {
    mbedtls_printf( " failed\r\n  ! mbedtls_ssl_setup returned -0x%x\r\n", -ret );
    goto exit;
  }
```

```
    mbedtls_printf( " ok\r\n" );

reset:
#ifdef MBEDTLS_ERROR_C
    if(ret != 0)
    {
        uint8_t error_buf[100];
        mbedtls_strerror( ret, (char *)error_buf, 100 );
        mbedtls_printf("Last error was: -0x%x - %s\r\n", -ret, error_buf );
    }
#endif

    mbedtls_net_free(&client_fd);

    mbedtls_ssl_session_reset(&ssl);

    /*
     * 5. Wait until a client connects
     */
    mbedtls_printf( "    . Waiting for a remote connection ...\r\n" );

    if((ret = mbedtls_net_accept(&listen_fd, &client_fd, NULL, 0, NULL)) != 0)
    {
        mbedtls_printf("    => connection failed\r\n    ! mbedtls_net_accept returned -0x%x\r\n", -ret);
        goto exit;
    }

    mbedtls_ssl_set_bio(&ssl, &client_fd, mbedtls_net_send, mbedtls_net_recv, NULL);

    mbedtls_printf("    => connection ok\r\n");

    /*
     * 6. Handshake
     */
    mbedtls_printf("    . Performing the SSL/TLS handshake...");

    while((ret = mbedtls_ssl_handshake(&ssl)) != 0)
    {
```

```
    if(ret != MBEDTLS_ERR_SSL_WANT_READ && ret !=
MBEDTLS_ERR_SSL_WANT_WRITE)
    {
      mbedtls_printf(" failed\r\n  ! mbedtls_ssl_handshake returned -0x%x\r\n", -ret);
      goto reset;
    }
  }

  mbedtls_printf(" ok\r\n");

  /*
   * 7. Read the HTTP Request
   */
  mbedtls_printf("  < Read from client:");
  do
  {
    len = sizeof(buf) - 1;
    memset(buf, 0, sizeof(buf));
    ret = mbedtls_ssl_read(&ssl, buf, len);

    if(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
MBEDTLS_ERR_SSL_WANT_WRITE)
     {
      continue;
    }
    if(ret <= 0)
    {
      switch(ret)
      {
        case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
          mbedtls_printf(" connection was closed gracefully\r\n");
          break;

        case MBEDTLS_ERR_NET_CONN_RESET:
          mbedtls_printf(" connection was reset by peer\r\n");
          break;

        default:
          mbedtls_printf(" mbedtls_ssl_read returned -0x%x\r\n", -ret);
```

```
                break;
            }

            break;
        }

        len = ret;
        mbedtls_printf(" %d bytes read\r\n%s", len, (char *) buf);

        if(ret > 0)
        {
            break;
        }
    } while(1);

    /*
     * 8. Write the 200 Response
     */
    mbedtls_printf( "    > Write to client:" );
    len = sprintf((char *) buf, HTTP_RESPONSE, mbedtls_ssl_get_ciphersuite(&ssl));

    while((ret = mbedtls_ssl_write(&ssl, buf, len)) <= 0)
    {
        if(ret == MBEDTLS_ERR_NET_CONN_RESET)
        {
            mbedtls_printf(" failed\r\n    ! peer closed the connection\r\n");
            goto reset;
        }

        if(ret != MBEDTLS_ERR_SSL_WANT_READ && ret !=
MBEDTLS_ERR_SSL_WANT_WRITE)
        {
            mbedtls_printf( " failed\r\n    ! mbedtls_ssl_write returned -0x%x\r\n", -ret );
            goto exit;
        }
    }

    len = ret;
```

```
    mbedtls_printf(" %d bytes written\r\n%s", len, (char *) buf);

    mbedtls_printf("    . Closing the connection...");

    while((ret = mbedtls_ssl_close_notify(&ssl)) < 0)
    {
        if(ret != MBEDTLS_ERR_SSL_WANT_READ && ret !=
MBEDTLS_ERR_SSL_WANT_WRITE)
        {
            mbedtls_printf( " failed\r\n    ! mbedtls_ssl_close_notify returned -0x%x\r\n", -ret );
            goto reset;
        }
    }

    mbedtls_printf( " ok\r\n" );

    ret = 0;
    goto reset;

exit:
    mbedtls_net_free( &client_fd );
    mbedtls_net_free( &listen_fd );

    mbedtls_x509_crt_free( &srvcert );
    mbedtls_pk_free( &pkey );
    mbedtls_ssl_free( &ssl );
    mbedtls_ssl_config_free( &conf );
#if defined(MBEDTLS_SSL_CACHE_C)
    mbedtls_ssl_cache_free( &cache );
#endif
    mbedtls_ctr_drbg_free( &ctr_drbg );
    mbedtls_entropy_free( &entropy );

    while(1);
}
```

2.3.1 Use OpenSSL to create self-signed certificate

In this example, we use self-signed certificate to establish TLS connection. The OpenSSL is used to issue the self-signed certificate. This section simply introduces how to install OpenSSL on Windows and Linux Ubuntu.

- Windows
  OpenSSL does not provide executable installation file. Click Git for Windows to install OpenSSL. After the installation is complete, the default path of executable file is “C:\Program Files\Git\usr\bin\openssl.exe”. Users can add the “C:\Program Files\Git\usr\bin” to the PATH variables, and then enter “openssl” directly to execute this tool.
- Ubuntu
  Issue the command below in the terminal.

```
sudo apt install openssl
```

Check the OpenSSL on PC, and follow the steps below to create a legal self-signed certificate.

1. Set up ssl.conf configuration file

```
[req]
prompt = no
default_md = sha256
default_bits = 2048
distinguished_name = dn
x509_extensions = v3_req

[dn]
C = KY
ST = Cayman Islands
L = George Town
O = ARTERY
OU = SW Department
emailAddress = admin@example.com
CN = localhost

[v3_req]
subjectAltName = @alt_names

[alt_names]
DNS.1 = *.localhost
DNS.2 = localhost
IP.1 = 172.31.96.101
```

The [dn] field (Distinguished Name) is relevant to the certificate, which can be modified by users. The “O” (Organization) is the company name, “OU” (Organization Unit) is the organization name, and the “CN” (Common Name) is the certificate description. These contents can be modified and even be set in Chinese. Note that the file must be in UTF-8 encoding and does not contain BOM characters.

The [alt_names] field is used to set the SSL certificate domain name. If the domain name is not set, the SSL certificate will be considered invalid by some browsers. The self-signed certificate is created for the purpose of development and testing; therefore, it is recommended to include the all local domain names (localhost) or the local area network IPs that may be used to facilitate subsequent remote online testing.

2. Open the terminal tool and switch to the directory that stores “ssl.conf”, then use the OpenSSL command to generate the self-signed certificate and the corresponding private key. Enter the following command to create the private key (server.key) and certificate file (server.crt).

```
openssl req -x509 -new -nodes -sha256 -utf8 -days 3650 -newkey rsa:2048 -keyout server.key -out server.crt -config ssl.conf
```

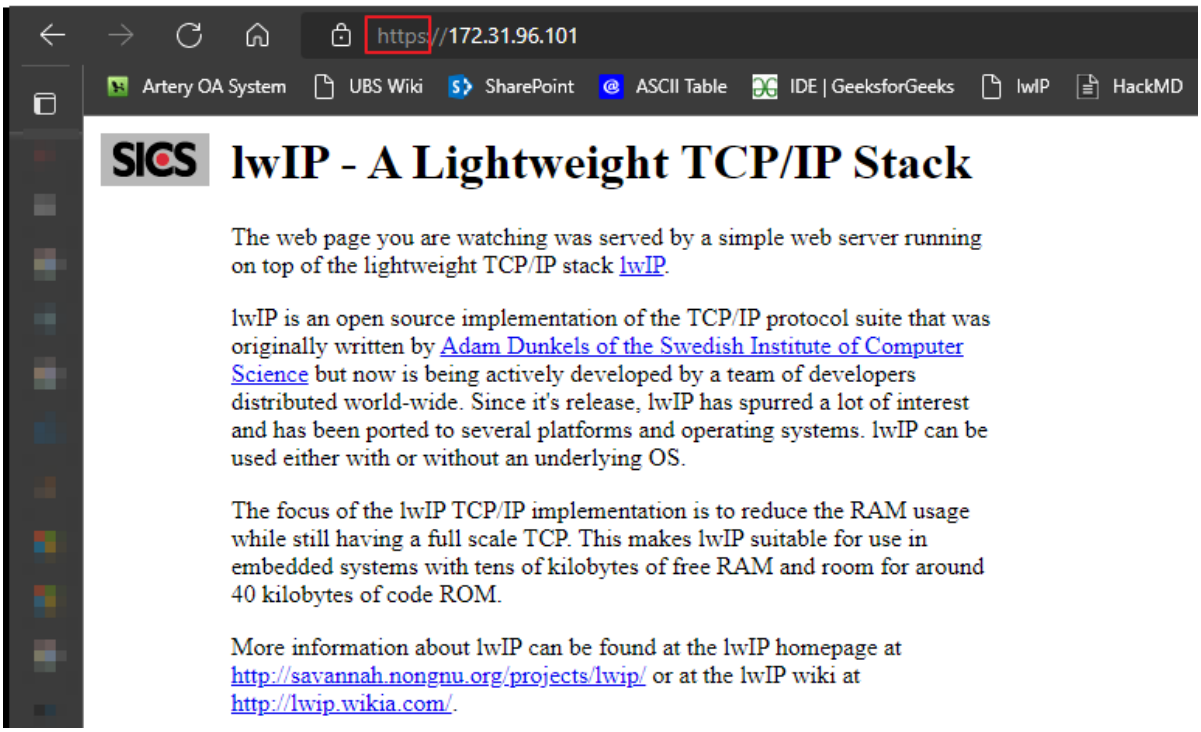
The generated private key and certificate file are stored in the directory of ssl.conf.

3. Import the self-signed certificate to the “trusted root certification authority”.
In addition to the self-signed certificate, web server (PKI infrastructure) also need to be set, and all endpoints that need to be securely connected should trust each other. Therefore, users need to install the self-signed certificate to the “trusted root certification authority”, so that the self-signed can be considered as “trusted online” by user’s operating system or browser. Follow the steps below to manually import the self-signed certificate.
   - Open the file Explorer and double click the *server.crt* file;
   - Click “Install certificate”;
   - Select “Current user” and click “Next”;
   - Click “Place all certificates in the following store” and click “Browse”;
   - Select “Trusted root certification authority” and click “OK”;
   - Click “Next”;
   - Click “OK”;
   - Click “Yes (Y)” in the security warning window to complete settings.
4. Import the private key and certificate to TLS server.
TLS server decrypts data. The TLS server is the MCU in this application note. Fill in the private key and certificate into the “mbedtls_test_srv_key” and “mbedtls_test_srv_crt” in the demo code.

2.4 Test result

- The web address of the browser starts with HTTPS, and the lock icon shows the locked state.

Figure 3. HTTPS web page



3 Revision history

Table 1. Document revision history

Date	Version	Revision note
2022.9.19	2.0.0	Initial release.
2023.8.8	2.0.1	Added FreeRTOS to manage tasks scheduling.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein.

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license grant by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement of any patent, copyright or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Aerospace applications or environment; (D) Weapons, and/or (E) Other applications that may cause injuries, deaths or property damages. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Resale of ARTERY products with provisions different from the statements and/or technical features stated in this document shall immediately void any warranty grant by ARTERY for ARTERY products or services described herein and shall not create or expand in any manner whatsoever, any liability of ARTERY.