Zephyr™ Project

Zephyr Authentication - **ZAUTH**

September 2020

More and more devices can now connect to each other, remote servers, and mobile devices. This added functionality provides end-users with better, more fully featured products, but at the very real risk of malicious attacks. The ability to authenticate peer communications, along with secure messaging, is how devices can protect themselves. Unfortunately, most RTOS systems do not have a framework or library to authenticate devices. The purpose of this ZAUTH proposal is to provide an easy to use framework to authenticate two peers over different transports such as Bluetooth and Serial. Zephyr is uniquely positioned with a large installed base and security as one of its key goals.

Some examples of device connectivity include:

a) Mobile app connecting to a room control via Bluetooth to control light settings.
b) Conveyor controller and remote sensors to detect speed and load.
c) Consumable authentication such as printer cartridges or continuous glucose monitor sensor.

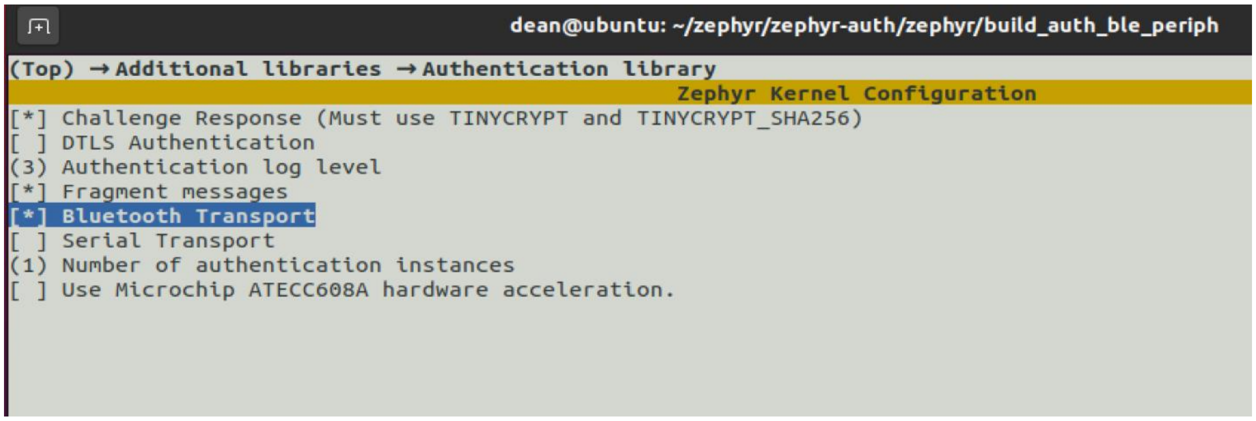
Proposed Change

ZAUTH is a library to provide a consistent set of APIs to authenticate peer devices over an arbitrary transport such as Bluetooth or Serial. The authentication method and transport layer are selectable using the existing Zephyr KConfig process. The underlying transport is abstracted from the ZAUTH authentication methods using an opaque transport handle. This initial proposal supports Bluetooth and Serial (UART) transports along with Challenge-Response and DTLS authentication methods. ZAUTH is design to accommodate additional authentication methods and transports.

ZAUTH library code is located in the following directories from the Zephyr root:

Directory	Description
`lib/auth`	Source and internal header files.
`include/auth`	Public header files for use by applications.
`samples/auth`	ZAUTH sample programs.
`test/lib/auth`	Unit test files.

Ease of use is an important goal of ZAUTH, most developers treat security as a headache. ZAUTH is easily integrated by using the following KConfig menu selections under "`Additional libraries->Authentication library`" as shown below.



```
dean@ubuntu: ~/zephyr/zephyr-auth/zephyr/build_auth_ble_periph
(Top) → Additional libraries → Authentication library
                                   Zephyr Kernel Configuration
[*] Challenge Response (Must use TINYCRYPT and TINYCRYPT_SHA256)
[ ] DTLS Authentication
(3) Authentication log level
[*] Fragment messages
[*] Bluetooth Transport
[ ] Serial Transport
(1) Number of authentication instances
[ ] Use Microchip ATECC608A hardware acceleration.
```

Authentication API

The Authentication API is designed to abstract away the authentication method and transport. The calling application configures the ZAUTH library, starts the authentication process and monitors results via a status callback. The API is also designed to handle multiple concurrent authentication processes, for example If device is acting as a Bluetooth Central and Peripheral. An example of the API used is shown in the following code snippet.

```
static authenticate_conn central_auth_conn;

void auth_status(struct authenticate_conn *auth_conn, enum auth_instance_id
instance, auth_status_t status, void *context)
{
    if(status == AUTH_STATUS_SUCCESSFUL) {
        printk("Authentication Successful.\n");
    } else {
        printk("Authentication status: %s\n", auth_lib_getstatus_str(status));
    }
}

/* BLE connection callback */
void connected(struct bt_conn *conn, u8_t err)
{
   /* start authentication */
   auth_lib_start(&central_auth_conn);
}


void main(void)
{
    int err = auth_lib_init(&central_auth_conn, AUTH_INST_1, auth_status, NULL,
                                opt_parms, flags);


    err = bt_enabled(NULL);
```

```
        while(true) {
            k_yield();
        }
    }
```

Client-Server Model

ZAUTH is designed as a client server model for the authentication message flow. The client initiates the authentication messaging sequence where the server responds. Depending on the authentication method chosen (Challenge-Response, DTLS, other), mutual authentication can be used to authenticate both sides of the connection. For some transports, this model maps nicely to the native transport model. Bluetooth is an example of this, a peripheral is in the server role and the central is in the client role. For Serial transports, the choice of which endpoint acts as the client or server is up to the application firmware.

Authentication Instances

Multiple authentication instances are possible concurrently authenticating connections over different communication links. For example, a Bluetooth central device could use different instances to authenticate different peripherals. Another example could be a HVAC controller with Bluetooth to communicate with mobile devices and a serial interface to control HVAC equipment. One instance would authenticate the mobile device, the second instance would authenticate the HVAC equipment.

Under the hood, an authentication Instance is a Zephyr thread and authentication method.

Authentication Methods

Two authentication methods are proposed, DTLS and simple Challenge-Response. However, the authentication architecture can support additional authentication methods in the future.

- DTLS. The TLS protocol is the gold standard of authentication and securing network communications. DTLS is part of the TLS protocol, but designed for IP datagrams which are lighter weight and ideal for resource constrained devices. Identities are verified using X.509 certificates and trusted root certificates. The DTLS handshake steps are used for authentication, a successful handshake means each side of the connection has been properly authenticated. A result of the DTLS handshake steps is a shared secret key which is then used to encrypted further communications. For the ZAUTH this key is not used, however it can be used for application level security (see MITM below).

- Challenge-Response. A simple Challenge-Response authentication method is an alternative lighter weight approach to authentication. This method uses a shared key and a random nonce. Each side exchanges SHA256 hash of Nonce and shared key, authentication is proven by each side knowing shared key. A Challenge-Response is not as secure and DTLS, however for some applications it is sufficient. For example, if a vendor wishes to restrict certain features of an IoT device to paid applications.

The proposed authentication is done at the application layer after connecting over the lower transport. This requires the firmware application to ignore or reject any messages until the authentication process has completed. This complicates the application firmware but does enable authentication independent of a vendor's stack such as Bluetooth, TCP/IP, or serial. In addition, most embedded engineers have no desire to modify a vendor's stack.

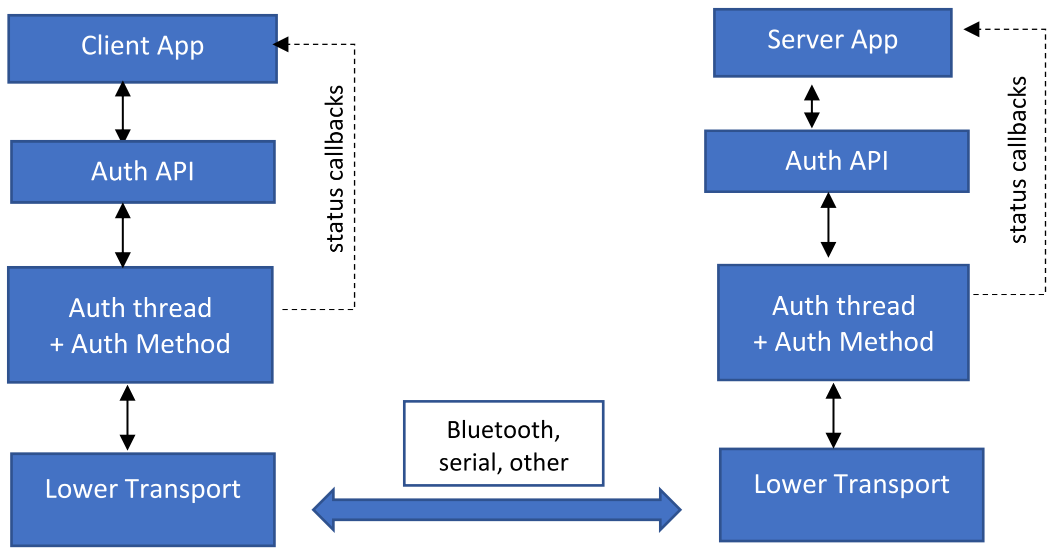
Secure Hardware

Secure hardware elements provide a high level of security and cryptographic functionality at a low cost and power consumption. Ideal for resource constraint IoT devices. The key material and operations are performed in hardware, no keys are stored in the code. Common use cases include ECC or RSA key generation, signature verification, certificate storage, and key storage. Each authentication method can potentially use secure hardware. For example, the Challenge-Response can use the secure hardware to store and verity the shared key. Some examples of secure hardware are the Microchip ATECC608A, NXP SE050, and Infineon OPTIGA Trust M SLS32AIA.

Initially ZAUTH will provide a secure hardware interface to the Challenge-Response authentication method. It is envisioned more secure hardware support will be added in later Zephyr releases.

Detail Design

The high-level diagram below shows the main ZAUTH components.

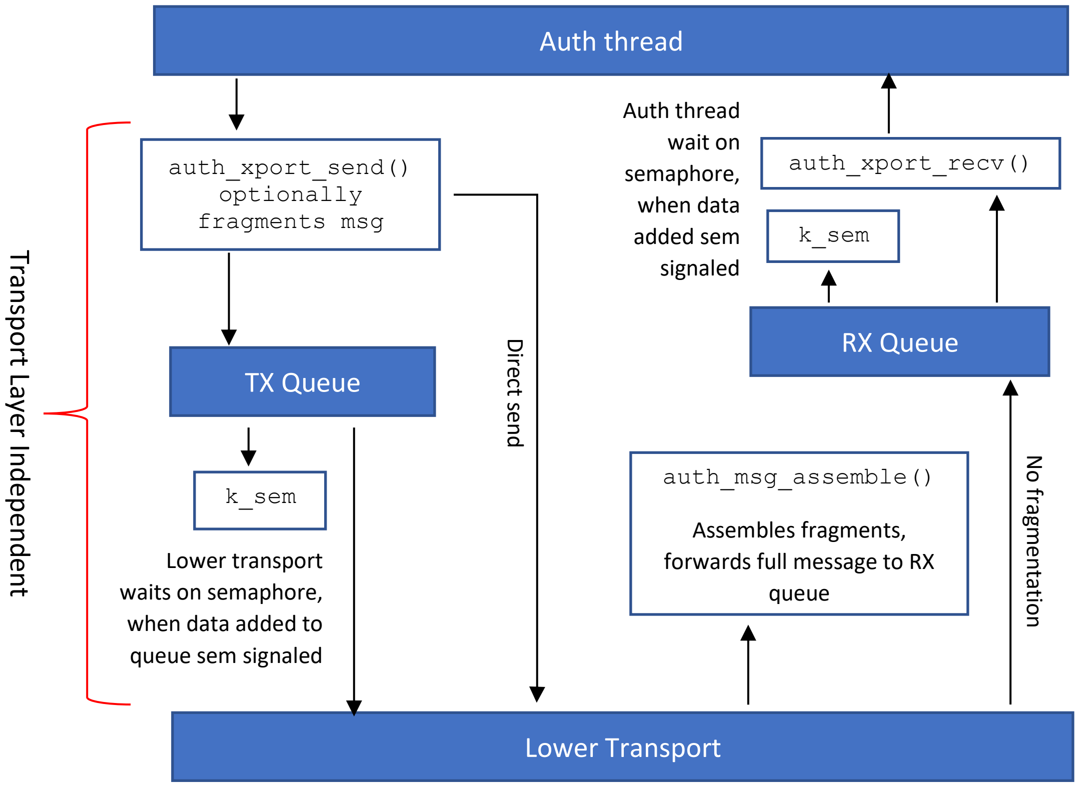


Authentication is performed in a separate thread started by the application.   Each authentication method uses a dedicated thread to exchange authentication message with their peer. Adding additional authentication methods is done by creating a authentication instance.   Messages are passed between the authentication thread and lower transport using an abstracted transport handle which maps to a Tx or Rx queue. The authentication threads are unaware of how messages are transferred. Optionally the lower transport can be configured to bypass the Tx queue and send the message directly to the lower transport, by passing the Tx queue. This is ideal for lower transports that handle their own message queueing.

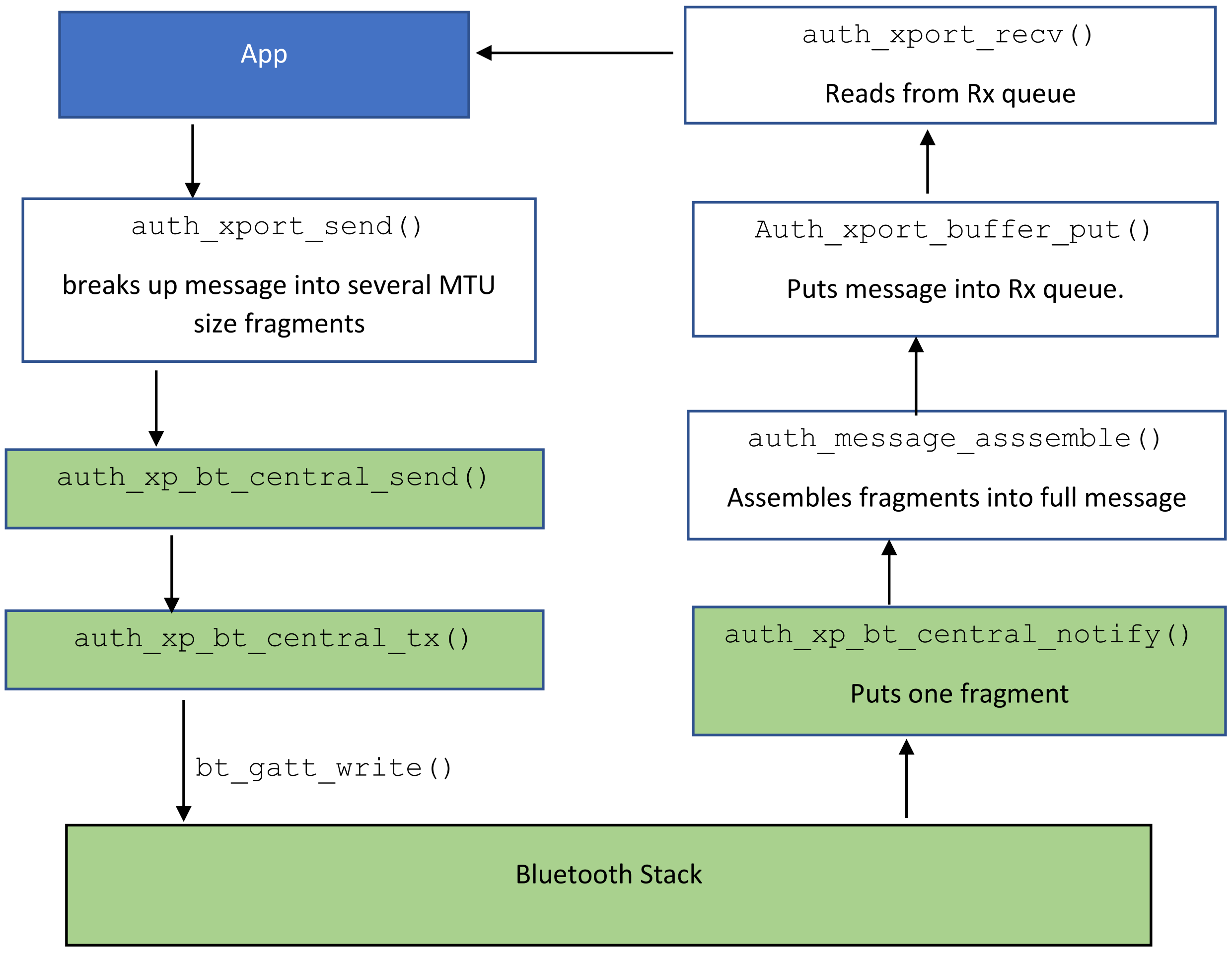
An Authentication method is a defined message protocol between two peers. The message protocol contains details of the contents and the order of messages. The DTLS protocol is an example of a detailed authentication protocol. Messages are different sizes and depending on the lower transport, may not fit into a transports MTU size. For example, the default MTU for Bluetooth is 23 bytes versus the 512 byte minimum possible for DTLS record.

Some authentication methods are designed to handle a continuous byte stream (i.e. TLS) others require complete messages (i.e. Challenge-Response). For those authentication methods requiring complete messages, ZAUTH can disassemble and re-assemble messages over the transport layer. For example, if a 267 byte message is send over a Bluetooth link with an MTU of 150, ZAUTH will break up the message into one 150 byte message and a second 117 byte fragments when sending. The receiving side will reassemble the fragments into the original 267 byte message before forwarding to the Rx queue.

The diagram below shows how the Tx and Rx queues are used along with message fragmentation.


Auth thread
auth_xport_send() optionally fragments msg
TX Queue
k_sem
Lower transport waits on semaphore, when data added to queue sem signaled
Direct send
Transport Layer Independent
Auth thread wait on semaphore, when data added sem signaled
auth_xport_recv()
k_sem
RX Queue
auth_msg_assemble()
Assembles fragments, forwards full message to RX queue
No fragmentation
Lower Transport

The Bluetooth Central Authentication sample (see samples/authentication/bluetooth/central_auth) provides a good example to drill deeper into the transport layer interface and how Bluetooth is "hooked up" to ZAUTH. The GREEN boxes are Bluetooth transport specific.

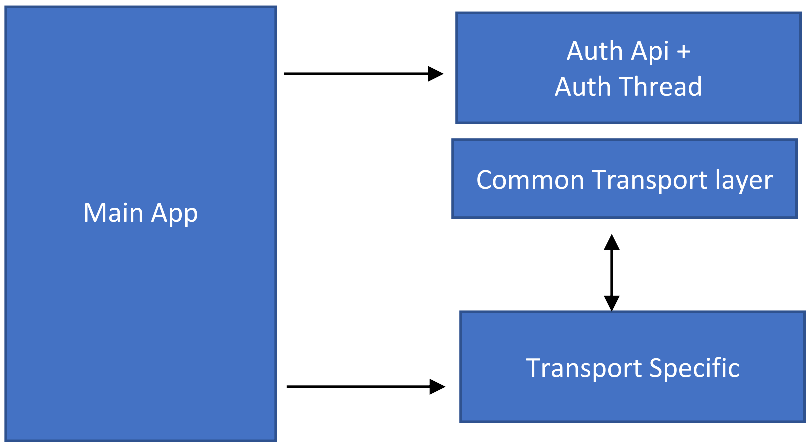


In `auth_xp_bt_init()` the Bluetooth connection (`struct bt_conn`) is added, along with the transport handle, to a connection using the `struct auth_xport_connection_map`

Transport Layer Interface

Transport layer details vary greatly, it does not make sense to create a one-size-fits-all transport API. ZAUTH separates the transport into transport independent and transport specific. For example, the details of the Bluetooth transport are in the `auth_xport_bt.c` file. This includes direct calls into the Zephyr Bluetooth stack. The transport common function, `auth_xport_init()`, calls the transport specific initialization function, passing the opaque transport handle (`auth_xport_hdl_t`) as an argument and transport specific parameters. The lower transport is responsible for mapping any transport specific variables to the transport handle. For example, the Bluetooth transport internally maps the transport handle to a Bluetooth connection handle, `struct bt_conn`.

The organization of the transport layers are show in the following diagram.



Additional Topics

Comparison of Bluetooth Pairing and Authentication

Simply put, for commercial IoT applications pairing is not authentication. The Bluetooth version 5 specification does define three methods of authentication (Bluetooth spec version 5, Vol 2, Part F, Section 4.2.9): a) Numeric Comparison, b) Passkey, and c) Out of Band. All of these methods require a display on the device and a human in the loop. In all of these options, the user is manually performing the authentication. For commercial IoT applications, this is not an option. Imagine a commercial building where hundreds of light switches (the IoT devices) must be manually paired. Error prone and costly work.

Pairing establishes an encryption key for both parties (Central and Peripheral), which is different from validating the identity of the remote peer. This is true for both Legacy and LE Secure pairing. While LE Secure pairing uses an ECC keypair and Diffie-Hellman Elliptic Curve to protect against eavesdropping, there is no capability to identify the remote peer.

GATT Authentication

GATT Authentication defined in the Bluetooth spec (Version 5, Vol 3, Part G, Section 8.1) is used to enable a per characteristic authentication using a shared key. It does not authenticate a Bluetooth peer (Central or Peripheral).

Google Fast Pair Service (GFPS)

This is a pairing method developed by Google to quickly pair consumer devices with and a phone and user's account. See: https://developers.google.com/nearby/fast-pair/spec, for details. It is part of Google's Nearby platform (see: https://developers.google.com/nearby). Authentication is accomplished by pre-shared keys provide by Google after registering your device (the thing you're developing) with Google. While GFPS provides a great user experience, it has several drawbacks, specifically: a) requires a mobile phone, b) requires Google approval for your device, and c) licensing may not be consistent with Zephyr.