silabs 21Q2 secure BLE device Security Lab User Manual

Contents

1 silabs 21Q2 secure BLE device Security Lab
2 BLE Security Lab Manual
3 Beyond the Basics
4 Secure Identity
5 Appendix A – I/O Capabilities and Pairing Methods Appendix B – Security Modes and Levels
6 Documents / Resources
6.1 References
7 Related Posts

silabs 21Q2 secure BLE device Security Lab

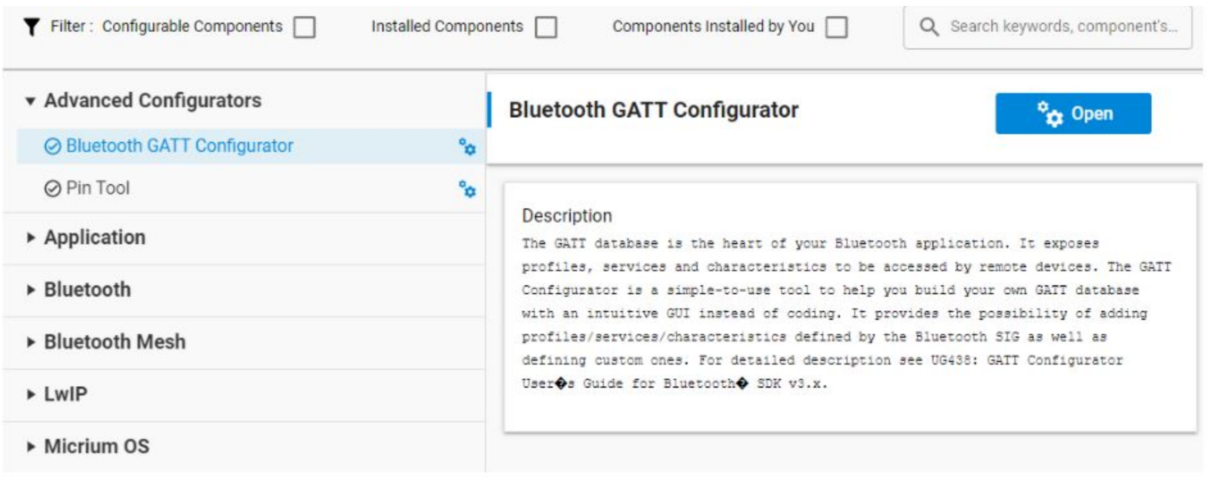
BLE Security Lab Manual

In this lab, you will see how to design a more secure BLE device. We will start with an overview of how to use some of the stack features and move on to some general advice about techniques for more secure connections and finally we will see how to use device certificates over BLE to identify a peripheral as authentic.

Getting Started

The Bluetooth sample application you will be building on is intended to be used with a bootloader. If you are working with a brand new EFR32MG21B, it will not have a bootloader. You can find a pre-built bootloader in the platform\bootloader\sample-apps\bootloader-storage-internalsingle\efr32mg21a010f1024im32-brd4181a folder of your SDK.

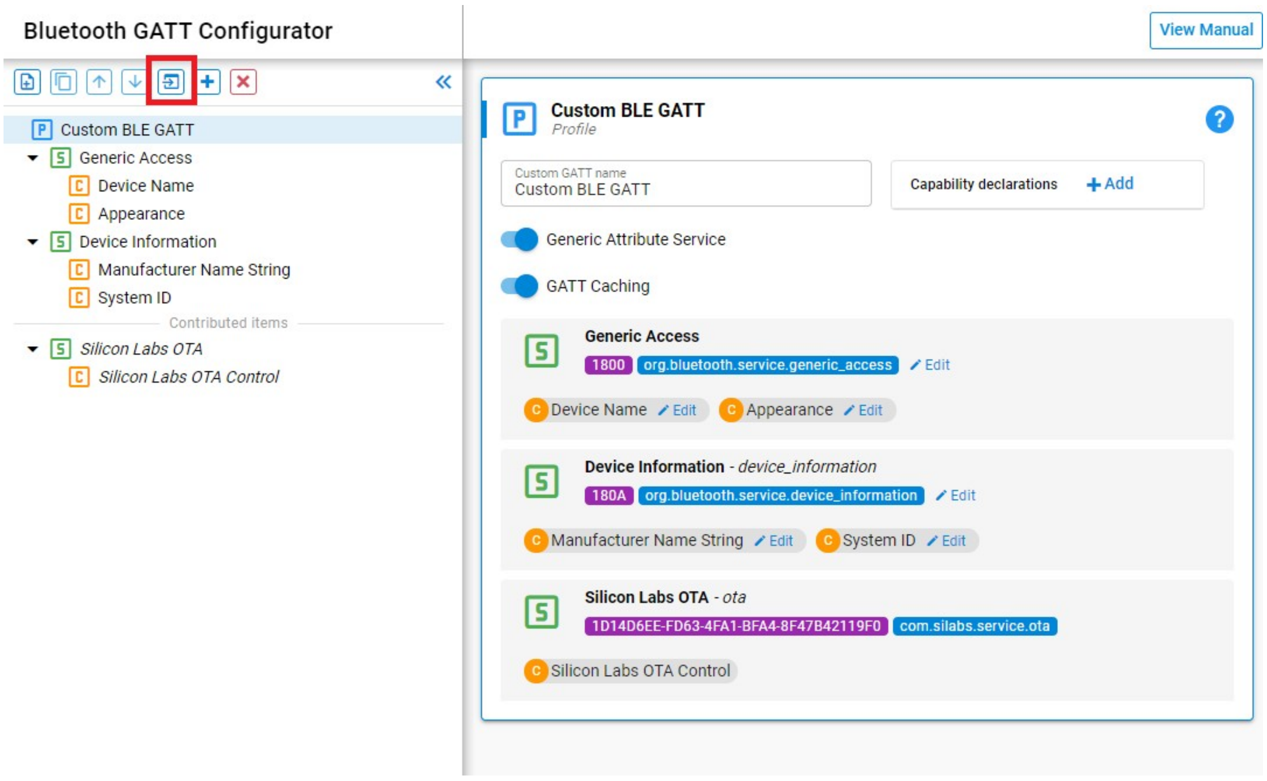
1. Start with a soc-empty sample app. This sample app is used as a template and makes a good starting point for any BLE application.
    1. Open the Silicon Labs Project Wizard from the Simplicity Studio File menu -> new.
    2. Select the BRD4181C and click the 'next' button.
    3. Click the 'Bluetooth (9)' checkbox under technology type.
    4. Highlight 'Bluetooth – SoC Empty' then click next.
    5. Click the 'Finish' button.
2. Now you can add some characteristics to see how protected and unprotected characteristics are treated differently.
    1. Open the project's slcp file by double-clicking it in the Project Explorer window
    2. Highlight the 'SOFTWARE COMPONENTS' tab and open the GATT configuration tool as shown below:


Filter : Configurable Components
Installed Components
Components Installed by You
Search keywords, component's...
Advanced Configurators
Bluetooth GATT Configurator
Pin Tool
Application
Bluetooth
Bluetooth Mesh
LwIP
Micrium OS
Bluetooth GATT Configurator
Open

Description

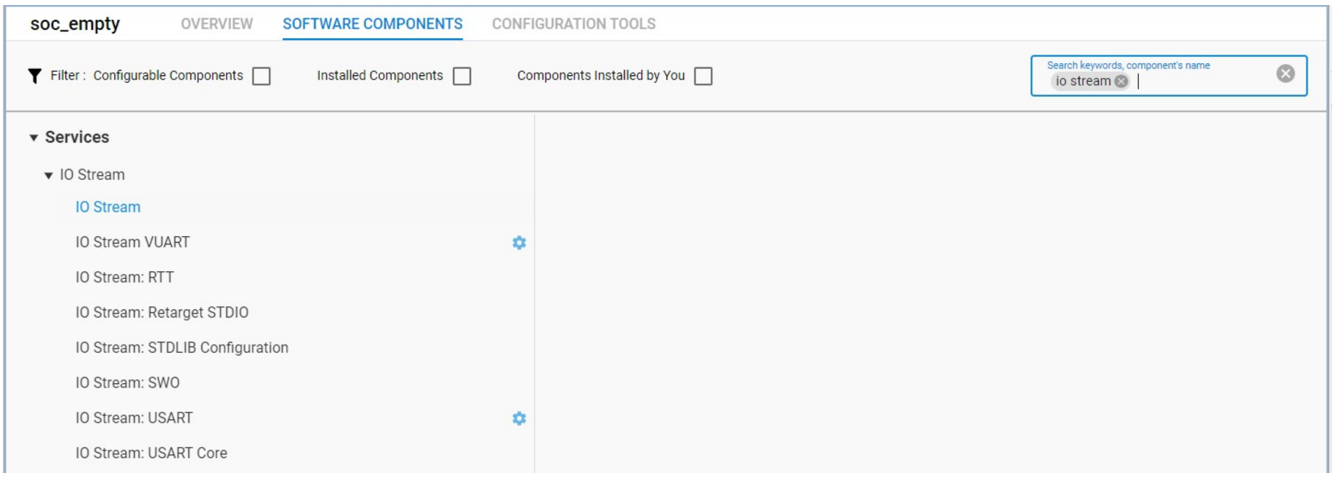
The GATT database is the heart of your Bluetooth application. It exposes profiles, services and characteristics to be accessed by remote devices. The GATT Configurator is a simple-to-use tool to help you build your own GATT database with an intuitive GUI instead of coding. It provides the possibility of adding profiles/services/characteristics defined by the Bluetooth SIG as well as defining custom ones. For detailed description see UG438: GATT Configurator User�s Guide for Bluetooth� SDK v3.x.

And use the import tool shown below to import the gatt_configuration.btconf file from the server folder in the provided materials.



The GATT database has a custom service, called ‘Training’, with some data that is protected and some that isn’t. This allows you to compare what happens when trying to access a protected characteristic vs an unprotected one. This is a quick way of making a device with very basic security.

3. We’ll use the serial port to print to the console in Simplicity Studio to track what’s going on in the application. The easiest way to find these components is by searching for them in the SOFTWARE COMPONENTS dialog as shown:



   1. Install the IO Stream USART component
   2. Install the IO Stream Retarget STDIO component
   3. Install the Standard I/O component
   4. Install the Log component
   5. Open the Board Control component and turn on the ‘Enable Virtual COM UART’
   6. Right-click the adapter in the ‘Debug adapters’ panel and select ‘Launch Console’. Select ‘Serial 1’ tab and place the cursor in the text entry field of the console window and press enter to wake up the console.

4. Create a local variable in sl_bt_on_event(), found in app.c, for saving the connection handle. The variable must be static since this function is called each time an event is raised by the stack and we want the value to be persistent. The connection handle will be used in a later

```
static uint8_t connection;
```

section of the lab.

5. Insert some app_log() statements for events to see when we're connected, security modes, etc
    1. Include the app_log.h header file

    ```
    #include "app_log.h"
    ```

    2. sl_bt_evt_connection_opened – print bond handle and save the connection handle. If the bond handle is 0xFF, no bond between the connected devices exists. Modify the existing event handler so that it looks something like this:

    ```
    case sl_bt_evt_connection_opened_id:
          app_log("connected with bond handle 0x%X\r\n", evt-
    >data.evt_connection_opened.bonding);
    connection = evt->data.evt_connection_opened.connection;
          break;
    ```

    3. sl_bt_evt_connection_parameters – security mode. This is done so that you can see when the security mode changes. There is a difference in the numbering of security modes where security mode 1, is enumerated with the value 0, etc. . Add the following event handler to your application:

    ```
    case sl_bt_evt_connection_parameters_id:
          app_log("parameters updated. Security mode %d\r\n",
    evt->data.evt_connection_parameters.security_mode +1);
          break;
    ```

    4. sl_bt_evt_connection_closed_id. This event handler is modified to update the connection handle. The value 0xFF is used to indicate that there is no active connection. The app_log() command is used to print out the reason for the connection being closed, the list of status codes is here. Modify the existing event handler so that it looks something like this:

    ```
    case sl_bt_evt_connection_closed_id:
                connection = 0xFF;
                app_log("connection closed, reason 0x%2X\r\n",
                        evt->data.evt_connection_closed.reason);

                // Restart advertising after client has disconnected.
                sc = sl_bt_advertiser_start(
                  advertising_set_handle,
                  advertiser_general_discoverable,
                  advertiser_connectable_scannable);
                app_assert_status(sc);
                break;
    ```

6. Build and flash the project. At this point, we will run the sample app to see how it behaves without any changes, besides the GATT database.
7. Connect with the EFRConnect mobile app as follows:
    1. Tap the 'Bluetooth Browser' icon.
    2. Tap the 'Connect' icon on the device named 'Training'.

8. Read the unprotected characteristic as follows:
    1. Tap the ‘More Info’ link under the unknown service with UUID a815944e-da1e-9d2a- 02e2-a8d15e2430a0.
    2. Read the unprotected characteristic, UUID f9e91a44-ca91-4aba-1c33-fd43ca270b4c by tapping the ‘Read’ icon. No surprises here. Since the characteristic is not protected in any way, it will be sent in plaintext.
9. Now read the protected characteristic, UUID d4261dbb-dcd0-daab-ec95-deec088d532b. Your mobile phone should prompt you to pair and connect, the message may vary depending on your mobile OS. After you accept the request to pair, you should a message on the console as follows:

    ```
    parameters updated. Security mode 2
    ```

    Note: Appendix A at the end of this manual has a summary of I/O capabilities and pairing methods for reference. Appendix B summarizes the Bluetooth security modes.

Security Manager Configuration

The security manager is part of the Bluetooth stack that determines which security features are used. These features include man-in-the-middle (MITM) protection, LE Secure connections (aka ECDH), requiring confirmation for bonding, etc. The security manager also handles I/O capabilities which are used to determine which method is used for pairing/bonding (see Appendix A for a summary). In this section you will see a simple setup.

1. Setup SM with desired configuration. The hardware for this lab makes it easy to display a passkey on the console. Passkey entry is a requirement to enable MITM protection. Add the following code to your sl_bt_system_boot_id event handler. This enables man-in-the-middle and informs the remote device that we have the ability to display a passkey, but that’s all.

    ```
    sl_bt_sm_configure(0x01,sl_bt_sm_io_capability_displayonly);
    ```

2. To display the passkey on the console, an event handler is required as shown below:

    ```
    case sl_bt_evt_sm_passkey_display_id:
          app_log("passkey: %d\r\n",
    evt->data.evt_sm_passkey_display.passkey);
          break;
    ```

3. Set the bonding mode, max number of bondings, etc. Use the following code to get started:

    ```
    sl_bt_sm_set_bondable_mode(0);

    sl_bt_sm_store_bonding_configuration(1,0);
    ```

    These settings can be used to limit an attacker’s ability to bond with your device. If your product only needs to have one user, then you could limit the maximum bonds to 1. A good place to add these calls is in the sl_bt_system_boot_id event handler. We won’t enable bonding at this time to make the rest of the lab go more smoothly but we do set a bonding policy to allow only one bond. For reference, the documentation for these APIs are found here and here .
4. Add event handlers for sl_bt_evt_sm_bonded_id and sl_bt_evt_sm_bonding_failed_id. The main use for these events is informative currently but later in the lab you will add functionality.

```
case sl_bt_evt_sm_bonded_id:
      app_log_info("new bond handle %d created\r\n", evt-
>data.evt_sm_bonded.bonding);
      break;

case sl_bt_evt_sm_bonding_failed_id:
      app_log_info("bonding failed, reason 0x%2X\r\n",
                   evt->data.evt_sm_bonding_failed.reason);
      break;
```

5. Build and flash to the target board. Connect with EFRConnect and read the protected characteristic as before. This time, you will see a passkey displayed on the console. Enter this passkey on your mobile phone when prompted.
6. Try out bonding confirmation. This feature gives the user the ability to require that bonding requests be confirmed. Doing so gives the application control over which peer devices it bonds with. One possibility is to require the user to press a button before allowing the bond.
    1. Open the Bluetooth settings in your mobile phone and remove the bond to the EFR32 device. Mobile phone implementations vary so this step may not be necessary. If you don't see the 'Training' device in your Bluetooth settings, just proceed to the next step.
    2. In software components, install one instance of the simple button handler.
    3. Include the header file sl_simple_button_instances.h in app.c

    ```
    #include "sl_simple_button_instances.h"
    ```

    4. Add a handler for the sl_bt_evt_sm_bonding_confirm_id event. The main job of this event handler is to inform the user that a remote device is requesting a new bond.

    ```
    case sl_bt_evt_sm_confirm_bonding_id:
          app_log_info("confirm bonding? PB0 for yes\r\n");
          break;
    ```

    5. Add a callback function for the simple button handler to send a signal to the Bluetooth stack indicating that a button has been pressed. This overrides the default callback which simply returns.

    ```
    void sl_button_on_change(const sl_button_t *handle)
    {
      if(handle== &sl_button_btn0 && sl_button_get_state(handle) == 0){
          sl_bt_external_signal(0x01);
      }
    }
    ```

    6. Add an external signal event handler. This event is raised in response to receiving a signal, such as in the previous step. The external signal event will be used to confirm bonding.

    ```
    case sl_bt_evt_system_external_signal_id:
              if(evt->data.evt_system_external_signal.extsignals & 0x01 &&
    connection != 0xFF){
                    app_log("confirming bonding for connection %d\r\n",
                            connection);
                    sl_bt_sm_bonding_confirm(connection,true);
              }
              break;
    ```

    7. Change the call to sl_bt_sm_configure to require bonding confirmation such as

```
sl_bt_sm_configure(0x08,sl_bt_sm_io_capability_displayonly);
```

8. Rebuild and flash.
9. Connect with EFRConnect and read the protected characteristic as before. Now you will see a message on the console as follows:

```
confirm bonding? PB0 for yes
```

   Press PB0 to confirm the bonding. Now the console will display the passkey to be entered on the mobile phone for bonding. Enter the passkey to complete the bonding process.

Tip: Use the default case in the event handler to print out a message when the stack sends an event that is not handled. The stack may be trying to tell you something important.

Beyond the Basics

At this point, you have taken advantage of the security features that our stack has to offer. Now let's improve the implementation through wise use of features at our disposal. The following steps are optional and independent of each other, you can build and flash after each one to see the behavior or try them all together.

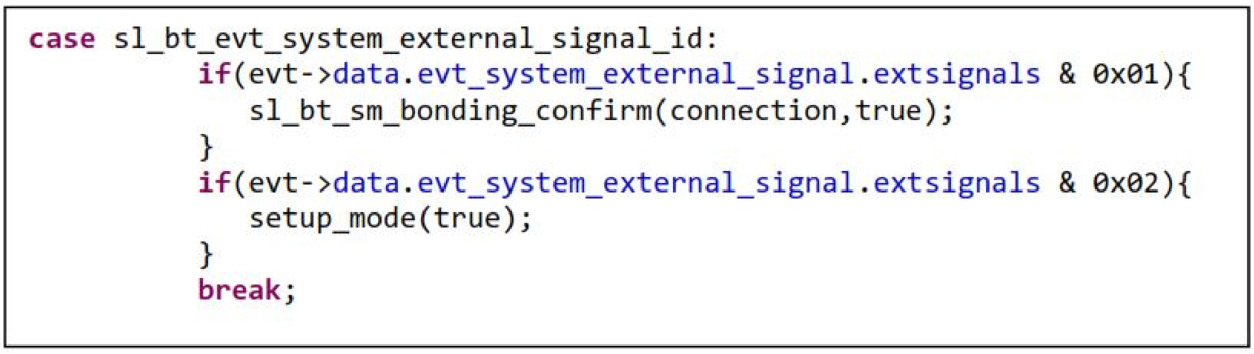
1. Disconnect on failed bond attempts. This is a good place to detect threats. If the remote device does not support encryption/authentication or just does not have the correct keys, it could be a hacker. So, let's break the connection. Try adding a call to sl_bt_connection_close() in the sl_bt_sm_bonding_failed_id event. The API is documented here.

```
sl_bt_connection_close(connection);
```

   You can test this feature by entering the wrong passkey.
2. Only allowing bonding at certain times. This limits the time that an attacker has to form a bond and makes it possible to use the 'only allow bonded connections' feature. The designer can choose how to enable or disable bondable mode. For demonstration purposes here, we'll enable a 'setup mode' with PB1 and use a timer to disable it after 30 seconds.
   1. Install a second instance of the simple button interface. This will enable the use of PB1.
   2. Modify the callback to send a different signal to the stack to enable/disable bonding. The result should look something like this:

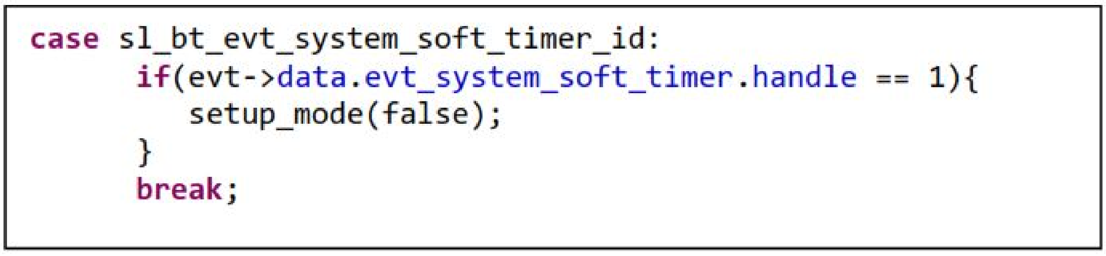
```
void sl_button_on_change(const sl_button_t *handle)
{
  if(handle== &sl_button_btn0 && sl_button_get_state(handle) == 0){
      sl_bt_external_signal(0x01);
  }
  else if(handle== &sl_button_btn1 && sl_button_get_state(handle) ==
0){
      sl_bt_external_signal(0x02);
  }
}
```

   3. Modify the external signal event handler so that it handles this new signal. The result should like this:



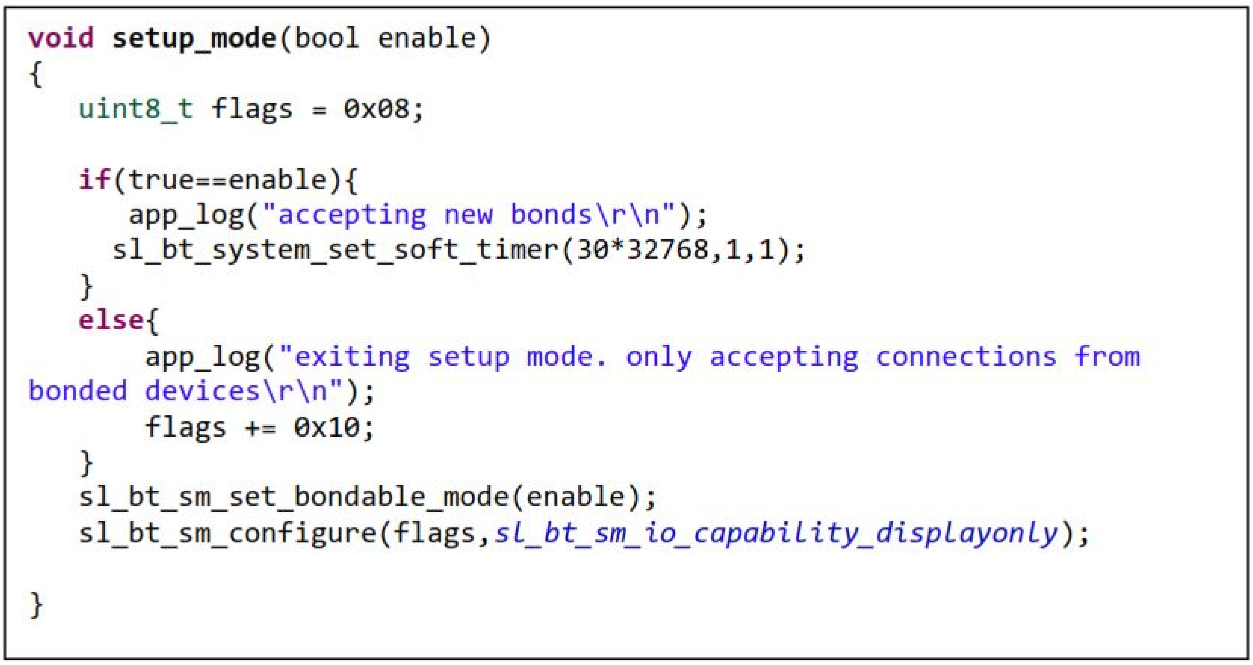
```
case sl_bt_evt_system_external_signal_id:
          if(evt->data.evt_system_external_signal.extsignals & 0x01){
             sl_bt_sm_bonding_confirm(connection,true);
          }
          if(evt->data.evt_system_external_signal.extsignals & 0x02){
             setup_mode(true);
          }
          break;
```

   4. Add an event handler for the sl_bt_evt_system_soft_timer_id event. This will used to disable setup mode.



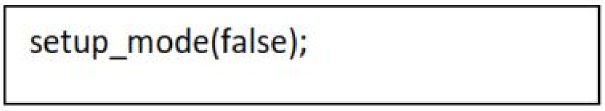
```
case sl_bt_evt_system_soft_timer_id:
      if(evt->data.evt_system_soft_timer.handle == 1){
         setup_mode(false);
      }
      break;
```

   5. The following code can be used to enable bondable mode and allow all connections or to disable bondable mode and only allow connections from bonded devices:



```
void setup_mode(bool enable)
{
   uint8_t flags = 0x08;

   if(true==enable){
      app_log("accepting new bonds\r\n");
     sl_bt_system_set_soft_timer(30*32768,1,1);
   }
   else{
        app_log("exiting setup mode. only accepting connections from
bonded devices\r\n");
        flags += 0x10;
   }
   sl_bt_sm_set_bondable_mode(enable);
   sl_bt_sm_configure(flags,sl_bt_sm_io_capability_displayonly);

}
```

   6. Add the following call in the sl_bt_system_boot_id event handler



```
setup_mode(false);
```

   7. Build the project and flash it to the device.
   8. Try connecting to the device with EFRConnect. The connection should fail.
   9. Now try pressing PB1 before connecting with EFRConnect. This time the connection will be successful. After 30 seconds you'll see a message on the console indicating that the device is exiting setup mode. This means that bondable mode is now disabled.

3. Increase security on forming a connection. Since security is optional, we should request an encrypted connection as soon as possible rather than relying on GATT characteristics. The API is documented here. A good place to call this API is in the sl_bt_evt_connection_opened_id event.The connection handle is available in the connection variable.

```
if(evt->data.evt_connection_opened.bonding != 0xFF){
        sl_bt_sm_increase_security(connection);
    }
```

Secure Identity

Now that we have a more secure Bluetooth device, lets improve the authentication step. You've already seen how to verify the secure identity of vault devices with the command line in previous training labs. In this section, we will see how one BLE device can verify the identity of another BLE device by requesting its certificate chain and sending a challenge. All secure vault parts hold their own device certificate and batch certificate. The factory and root certificates are hard coded into the client application to enable verification of the entire certificate chain. Refer to AN1268 for more details on secure identity.

1. Define a global buffer for storing the device attestation signature as below:

```
static uint8_t device_attestation_signature[64];
```

2. Set the security manager configuration to use JustWorks pairing. This is done so that the connection is encrypted. In practice, MITM protection should be used but to keep the lab simple, we will use JustWorks. Change the call to sl_bt_sm_configure back to the following:

```
sl_bt_sm_configure(0x00,sl_bt_sm_io_capability_displayonly);
```

   Also, comment out the call to setup_mode(true) in the system_boot event handler.

3. Open helpers.c from the provided materials and copy the contents into app.c. These callback functions perform tasks such as segmenting the certificates so that they can be sent over BLE, verifying the certificate chain, and generating/verifying the challenge.
4. It is necessary to determine the maximum transfer unit (MTU) size so that certificates can be segmented and reassembled. Define a global variable to save the MTU as shown here:

```
static uint16_t mtu;
```

   Then add an event handler for the GATT MTU exchanged event as shown below:

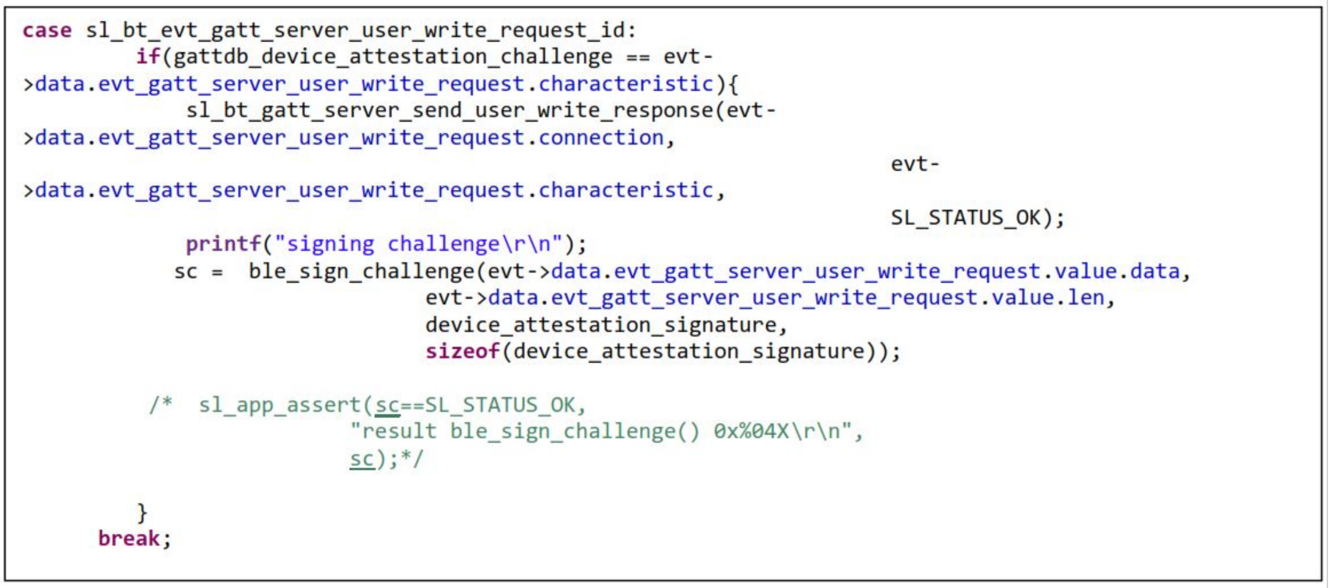
```
case sl_bt_evt_gatt_mtu_exchanged_id:
    mtu = evt->data.evt_gatt_mtu_exchanged.mtu;
    break;
```

5. There are three user data characteristics which can be read. These characteristics are used to communicate the device certificate, batch certificate and the challenge. A callback function is used to handle these user read requests. Add a handler to call this function as shown below:

```
case sl_bt_evt_gatt_server_user_read_request_id:
    user_read_request_cb(&evt->data.evt_gatt_server_user_read_request);
   break;
```

   The callback uses the MTU from step #2 to segment and send the certificates as needed. It also handles sending the signed challenge.

6. The client sends a challenge, a random number to be signed by the server, by writing one of the GATT characteristics. For this reason, the application needs to have a handler for the user write request event as below:



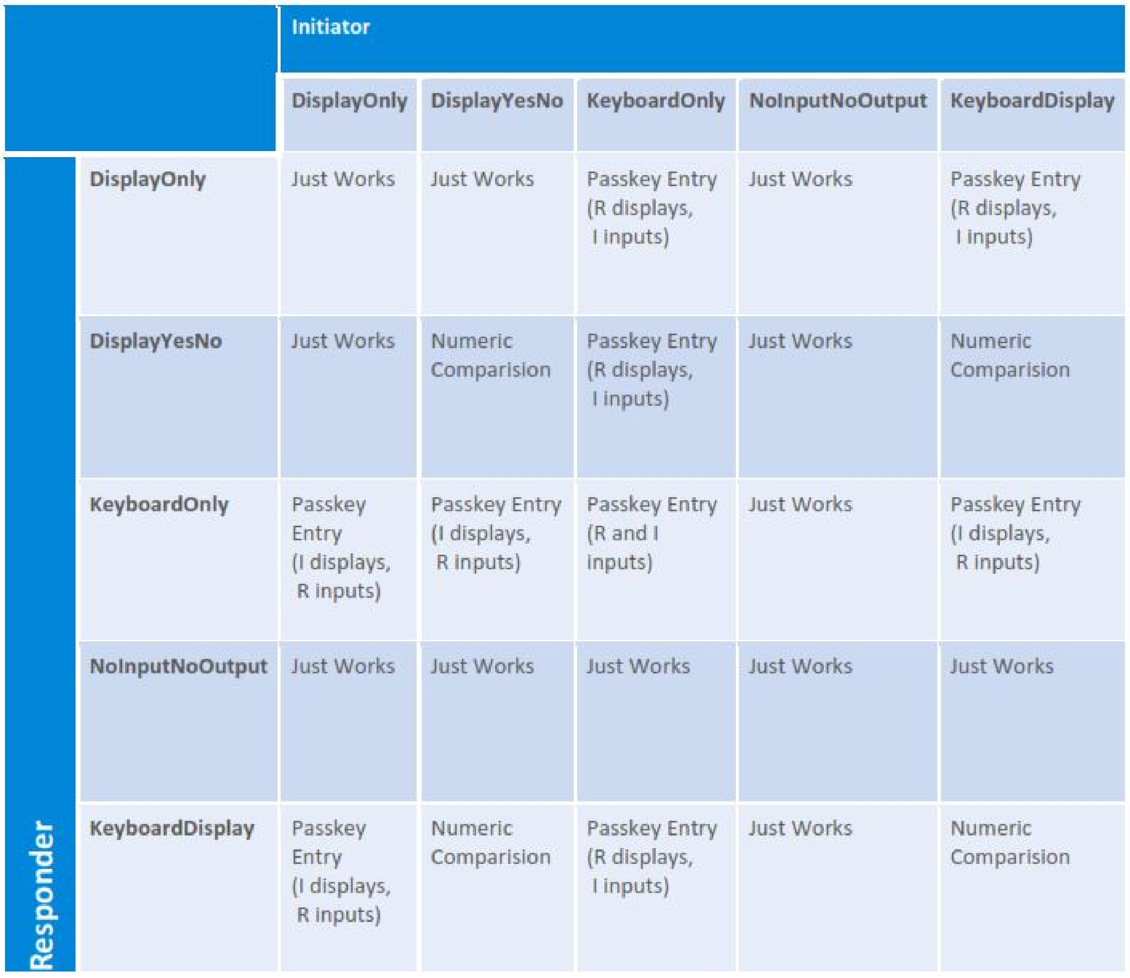
```
case sl_bt_evt_gatt_server_user_write_request_id:
        if(gattdb_device_attestation_challenge == evt-
>data.evt_gatt_server_user_write_request.characteristic){
            sl_bt_gatt_server_send_user_write_response(evt-
>data.evt_gatt_server_user_write_request.connection,
                                                                        evt-
>data.evt_gatt_server_user_write_request.characteristic,
                                                                        SL_STATUS_OK);
            printf("signing challenge\r\n");
           sc =  ble_sign_challenge(evt->data.evt_gatt_server_user_write_request.value.data,
                                    evt->data.evt_gatt_server_user_write_request.value.len,
                                    device_attestation_signature,
                                    sizeof(device_attestation_signature));

         /*  sl_app_assert(sc==SL_STATUS_OK,
                          "result ble_sign_challenge() 0x%04X\r\n",
                          sc);*/

        }
      break;
```

7. Add secure identity support files to the project:
   1. app_se_manager_macro.h, app_se_manager_secure_identity.c and app_se_secure_identity.h from the provided materials to the project. These files contain some helper functions for tasks such as getting the size of the certificate, getting the device public key and signing a challenge.
   2. Include app_se_manager_secure_identity.h in app.c.
8. Import the provided gatt_configuration-attest.btconf from the provided materials. This GATT database called secure attestation which includes four characteristics which will be used to verify the identity of our device. These include the device certificate, batch certificate, challenge and response.
9. The client, which is used to simulate a device such as gateway, is provided as a complete project since it is more complex to build. In general, the operation of the client is as follows:
   1. Scans for devices advertising the secure attestation service and connects to them.
   2. Discovers the GATT database services and characteristics.
   3. Reads the device and batch certificates and verifies the certificate chain using the factory and root certificate which it has stored in flash.
   4. Sends a random challenge to the server.
   5. Attempts to verify the response to the challenge.
   6. Closes the connection if either verification fails.
10. Build and flash the server project to your server WSTK /radioboard.
11. Import the client project from the client folder in the provided materials. Build and flash the client project to your client WSTK/radioboard.
12. Press reset on the client WSTK and open the serial console. The client begins scanning for devices advertising our secure identity service and will connect when it finds one.
13. The client will display some messages to indicate that it has found the server with the desired service and status messages about the verification of the certificate chain.
14. If the verification passes, the client will generate a random number, called a challenge, and send it to the server. The server will sign the challenge with its securely held private device key and the signature back to the client, this is called a challenge response. The client then uses the public key in the previously received device certificate to verify the signature. This is done to confirm that the server really has the private key that it

claimed to have. If the challenge is verified correctly, a message is displayed to that effect; otherwise, the connection is closed, and a message is displayed explaining why.

15. Now send an invalid certificate to confirm that the verification really works. You can modify user_read_request_cb() to corrupt either the certificate data or the challenge response.

Appendix A – I/O Capabilities and Pairing Methods



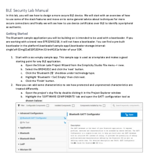
		Initiator				
		DisplayOnly	DisplayYesNo	KeyboardOnly	NoInputNoOutput	KeyboardDisplay
Responder	DisplayOnly	Just Works	Just Works	Passkey Entry (R displays, I inputs)	Just Works	Passkey Entry (R displays, I inputs)
	DisplayYesNo	Just Works	Numeric Comparision	Passkey Entry (R displays, I inputs)	Just Works	Numeric Comparision
	KeyboardOnly	Passkey Entry (I displays, R inputs)	Passkey Entry (I displays, R inputs)	Passkey Entry (R and I inputs)	Just Works	Passkey Entry (I displays, R inputs)
	NoInputNoOutput	Just Works	Just Works	Just Works	Just Works	Just Works
	KeyboardDisplay	Passkey Entry (I displays, R inputs)	Numeric Comparision	Passkey Entry (R displays, I inputs)	Just Works	Numeric Comparision

Appendix B – Security Modes and Levels

Security mode 1 is the only mode supported for Bluetooth Low Energy in the Silicon Labs' stack. The levels are as follows:

- Level 1 no security
- Level 2 unauthenticated pairing with encryption
- Level 3 authenticated pairing with encryption
- Level 4 authenticated secure connections with strong encryption (ECDH key exchange)

Documents / Resources

	**silabs 21Q2 secure BLE device Security Lab** [pdf] User Manual 21Q2 secure BLE device Security Lab, secure BLE device Security Lab, Security Lab

BLE Security Lab Manual

In this lab, you will see how to design a more secure BLE device. We will start with an overview of how to use some of the stack features and move on to some general advice about techniques for more secure connections and finally we will see how to use device certificates over BLE to identify a peripheral as authentic.

Getting Started

The Bluetooth sample application you will be building on is intended to be used with a bootloader. If you are working with a brand new EFR32MG21B, it will not have a bootloader. You can find a pre-built bootloader in the platform\bootloader\sample-apps\bootloader-storage-internal-single\efr32mg21a010f1024im32-brd4181a folder of your SDK.

1. Start with a soc-empty sample app. This sample app is used as a template and makes a good starting point for any BLE application.
    a. Open the Silicon Labs Project Wizard from the Simplicity Studio File menu -> new.
    b. Select the BRD4181C and click the 'next' button.
    c. Click the 'Bluetooth (9)' checkbox under technology type.
    d. Highlight 'Bluetooth – SoC Empty' then click next.
    e. Click the 'Finish' button.
2. Now you can add some characteristics to see how protected and unprotected characteristics are treated differently.
    a. Open the project's slcp file by double-clicking it in the Project Explorer window
    b. Highlight the 'SOFTWARE COMPONENTS' tab and open the GATT configuration tool as shown below:

References

- **Status Codes Silicon Labs**