Developer Study Guide: Bluetooth® Low Energy Security

Release	:	1.1.3
Document Version	:	1.1.3
Last updated	:	5th July 2021

Contents

Revision History

Version	Date	Author	Changes
1.0.0	3rd October 2019	Martin Woolley, Bluetooth SIG	Initial version
1.1.0	6th March 2020	Martin Woolley, Bluetooth SIG	Core specification references update to use version 5.2. New section covering security mode 3, used with broadcast isochronous channels, added.
1.1.2	18th November 2020	Martin Woolley, Bluetooth SIG	Language changes and brand alignment.
1.1.3	5th July 2021	Martin Woolley, Bluetooth SIG	**Release**: Zephyr code now based on version 2.6.0 of the Zephyr SDK **Document**: Code fragments revised to be based on Zephyr 2.6.0

1. Introduction

About This Study Guide

This is an educational resource aimed at software developers and people in other technical roles that work with *Bluetooth®* Low Energy. It is concerned with the security capabilities of Bluetooth Low Energy (LE) when used with the Generic Access Profile (GAP) and Generic Attribute Profile (GATT), as shown in Figure 1. Such devices are sometimes referred to as Bluetooth *connectable devices*. In this guide, the term *Bluetooth LE* will be used as a shorthand for *Bluetooth LE with GATT and GAP*.

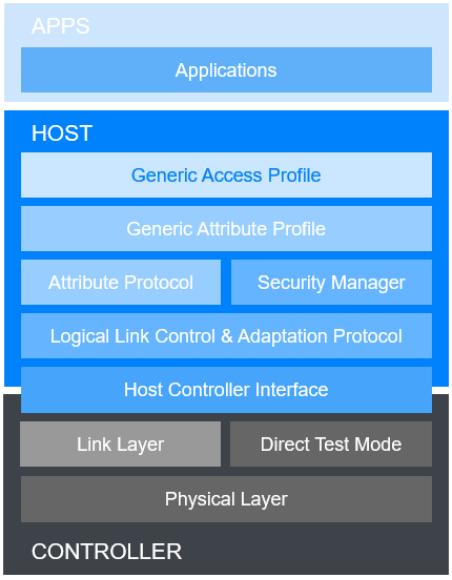


Figure 1 - The Bluetooth LE stack with GATT and GAP

General security concepts and issues will be discussed, and the security features and capabilities of Bluetooth LE will be explained. In some cases, you will have the opportunity to implement certain security features in code so that some hands-on experience is gained in addition to understanding of the theory. These hands-on exercises are completely optional.

The Bluetooth Core Specification is the primary reference for Bluetooth technical details, including security. This study guide is intended to make the learning curve less steep for people who need to understand what security features Bluetooth LE has, how to use them, and how they work. The descriptions of security procedures in this study guide have sometimes been intentionally simplified to aid understanding. Error conditions and error handling might not be mentioned, for example.

The Bluetooth Core Specification should always be consulted for precise reference information on Bluetooth security features, and you are encouraged to consult the Bluetooth Core Specification while reading this resource. References to relevant sections of the Bluetooth Core Specification version 5.2 are given throughout.

This is an educational resource and is **not** a substitute for the Bluetooth specifications.

If you are unfamiliar with terms like GATT and GAP, you should read the *LE basic Theory* document (part of the Bluetooth LE Developer Study Guide) before proceeding further with this study guide.

Bluetooth mesh networking uses the Bluetooth LE controller for radio communication. However, its security features are different to those used with the Bluetooth LE connectable devices, which this document covers. If you are interested in understanding Bluetooth mesh networking, you should review other resources from the Bluetooth Special Interest Group (SIG), including Bluetooth Mesh Networking - An Introduction for Developers and the Bluetooth Mesh Developer Study Guide.

If you wish to perform a rigorous appraisal of Bluetooth LE security, including scrutinising the cryptographic services involved, consult the Bluetooth Core Specification. This resource may provide useful and provide initial orientation for such purposes, but no more than that.

Software and Hardware Requirements

The hands-on coding exercises make use of the open-source Zephyr RTOS and SDK (software developer kit). It is assumed that you are able to use the Zephyr SDK to build and install binaries on a supported board. The Zephyr web site has documentation to help you get started.

You should be able to use any hardware supported by Zephyr, includes Bluetooth LE, and has at least 32kb memory. This information is provided to help you make informed decisions about developer boards that might be suitable.

You will also need an Android or iOS smartphone or tablet that supports Bluetooth LE version 4.2 or later. You should install the nRF Connect application by Nordic Semiconductor or an alternative application with similar capabilities.

A protocol analyser or *sniffer* is also useful but not mandatory.

About Security

Security is often cited as one of the most critical challenges facing technology professionals. Phenomena like the Internet of Things (IoT) and the increasing prevalence of connected devices and systems continues to focus minds on the subject of security.

Discussions about security often refer to the topic in binary terms. Devices or systems are deemed *secure* or *not secure*. But this is not the best way to think about security when designing and building products. It's more appropriate and useful to think about security as a continuum with no explicit security measures used at one end of the spectrum and every conceivable security measure taken at the other end. Somewhere in that continuum from *no security* to *very high security* is the level and type of security your product needs. Less would render your product vulnerable to one or more threats. More may mean higher development costs, higher hardware costs, degraded run-time performance and/or a degraded user experience (UX).

Security Requirements

Finding the right point on the security continuum for a product is something that only product designers, their colleagues, and other experts can do. It requires an understanding of the

functionality of the product, how it is used in practice, and who is using it and under what circumstances. A series of questions must be answered: What information is being processed and communicated? What is the value of that information and what are the consequences of an unauthorized party gaining access to that information? What are the likely consequences of data being tampered with and this going undetected? Must the user's privacy be safeguarded in some way? Are there any regulatory issues that apply to this product, and what security requirements do those regulations stipulate?

These are just a few examples of the questions that product teams must explore and answer in order that the security requirements of a product can be established.

Security Is Not Just About Communication

The Bluetooth specifications cover the subject of security as it applies to Bluetooth wireless communication. There are many other product security issues that fall outside of the world of communication, however. Where and how should security keys be stored? Must the use of a product be restricted to a specific user? How will that user be identified and authenticated? These are examples of questions whose answers will not be found within the Bluetooth specifications, and they will not be addressed here either.

Even if you implement all available Bluetooth security measures, this will not necessarily result in all your product's security requirements being met. The security requirements that Bluetooth security can address are only a subset of the full set of possible security requirements that a product may have, and product designers must ensure a sufficiently thorough and holistic assessment of product security requirements and solutions has taken place.

A Security Toolbox

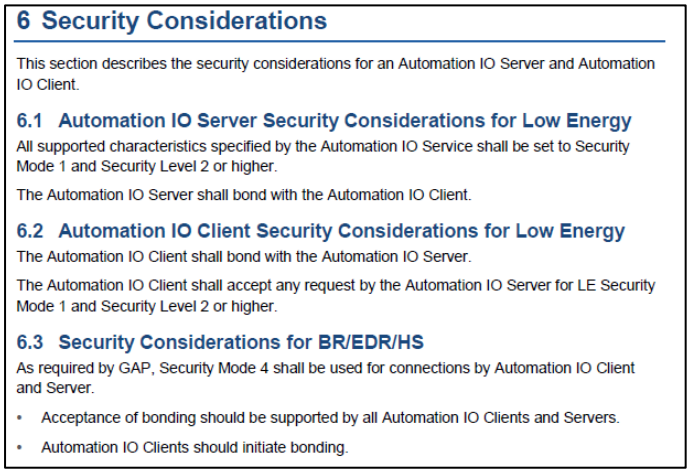
Bluetooth LE provides a collection of security capabilities and features, most of which are optional. You should think of this as a *toolbox* containing security tools with which to address specific security issues and meet specific security requirements. It is the responsibility of the product team, having ascertained the security requirements for a product, to meet those requirements. Where appropriate, this should be achieved through the use of selected Bluetooth LE security features.

The Bluetooth® Specifications

The Bluetooth Core Specification defines the security features and capabilities of the core Bluetooth stack. This specification is largely organized by stack layer, and security features appear in many of the layers. A quick search of the Bluetooth Core Specification for the word *security* will reveal that security is discussed in numerous parts of the core specification. Familiarity with the Bluetooth Core Specification is essential in gaining a complete understanding of Bluetooth security, and the core specification must always be considered the **primary** source of defining technical details. This study guide should be regarded as a complimentary resource, intended to help ease the ascent of the associated learning curve.

Bluetooth profile specifications define how Bluetooth can be used for a particular product type or application. This includes stating security requirements and security considerations. Figure 2 shows an excerpt from one of the standard Bluetooth profile specifications.

Custom profiles, such as those that incorporate custom-designed GATT services, are permitted and therefore, the profile's designer must assess and specify the applicable security requirements in such cases.



6 Security Considerations

This section describes the security considerations for an Automation IO Server and Automation IO Client.

6.1 Automation IO Server Security Considerations for Low Energy

All supported characteristics specified by the Automation IO Service shall be set to Security Mode 1 and Security Level 2 or higher.

The Automation IO Server shall bond with the Automation IO Client.

6.2 Automation IO Client Security Considerations for Low Energy

The Automation IO Client shall bond with the Automation IO Server.

The Automation IO Client shall accept any request by the Automation IO Server for LE Security Mode 1 and Security Level 2 or higher.

6.3 Security Considerations for BR/EDR/HS

As required by GAP, Security Mode 4 shall be used for connections by Automation IO Client and Server.

- Acceptance of bonding should be supported by all Automation IO Clients and Servers.
- Automation IO Clients should initiate bonding.

Figure 2 - Security considerations in the Automation IO Profile

There are a variety of test documents that list Bluetooth capabilities and cross-reference associated test cases that must be executed and passed to qualify a Bluetooth product or design. Some of the test cases relate to certain aspects of Bluetooth security. There is, for example, a test suite document specifically for the Security Manager Protocol. Its purpose is to allow verification of an implementation of that protocol. Test documents are a useful source of information on Bluetooth security.

Bluetooth Core Specification	https://www.bluetooth.com/specifications/bluetooth-core-specification/
GATT Profiles	https://www.bluetooth.com/specifications/gatt/
Test Documents	https://www.bluetooth.com/specifications/qualification-test-requirements/

2. Summary Recommendations

This short section makes some basic recommendations regarding security in Bluetooth LE connectable devices. Much of what follows will be found repeated in context elsewhere in this document. This summary is provided for convenience.

A recommended philosophical stance is to use those features that offer the best possible security, *unless there are good or insurmountable reasons why this cannot be*. If issues do exist that prevent you from adopting the theoretical ideal security design for your product, then you should make the minimum changes until your constraints are satisfied and your product is as secure as it can be under those limiting circumstances.

1. Always assess and document the security requirements of your product as an explicit step in any product design project.

2. Base your product on the latest Bluetooth Core Specification.

3. Ensure your product or application meets the security requirements specified in applicable profiles *at a minimum*.

4. Use LE Security Mode 1 Level 4.

5. Use a pairing association model that includes Man-in-the-Middle (MITM) protection. Design your product so that it has I/O capabilities that make this possible.

6. If the use of LE Legacy Pairing is unavoidable, it is recommended that a secure OOB (out-of-band) mechanism is used for authentication.

7. Protect the privacy of users through the use of private resolvable addresses.

8. Protect each attribute in your device's attribute table with access, encryption, and authentication permissions.

9. Verify through review and testing that your product meets the security requirements of applicable specifications, including those you defined for custom aspects of the product.

10. Check the security of your implementation through security testing and techniques like s security auditing.

3. Security Concepts

Before proceeding further, you should have an understanding of a number of fundamental security concepts and some terminology. This section introduces key topics and sometimes references external resources for formal definitions. There are a number of respected sources of security-related definitions on the internet. Where possible, the glossary provided by the National Institute of Standards and Technology (NIST) has been used. See https://csrc.nist.gov/glossary.

3.1 Security Issues

Attacker

A party who acts with malicious intent to compromise an information system – *see* https://csrc.nist.gov/glossary/term/attacker.

Confidentiality

I want to tell you something that is top secret, but we are in a crowded room!

When data is communicated between a number of intended, authorized devices using radio transmissions, another unauthorized device that is listening to the same radio frequency or frequencies can also receive the transmitted data. Listening to communication taking place between other devices in this way is known as *eavesdropping*. Any security requirement that stipulates this should not be allowed is said to be concerned with assuring the *confidentiality* of data communication between devices.

Confidentiality requirements are typically met through the use of *data encryption* (see below).

Authentication

Who are you, and how do I know you are who you say you are?

In everyday life, people are often required to present special documents, such as passports, to prove their identity.

In the context of computer-based communication systems, authentication is concerned with obtaining proof that a device you think you are communicating with really is that device and not some other, imposter device. Sometimes, you want to take the idea further and verify that the *person* using the device you are communicating with is who they say they are.

A specific attack that may become possible when authentication measures are absent is called the Man-in-the Middle (MITM) attack. This involves an attacker inserting themselves in between a conversation taking place between Alice and Bob. Alice thinks she is talking to Bob. Bob thinks he's talking to Alice. But, in reality, they are both talking to the MITM attacker who is relaying data between Alice and Bob, stealing data or manipulating it in the process.

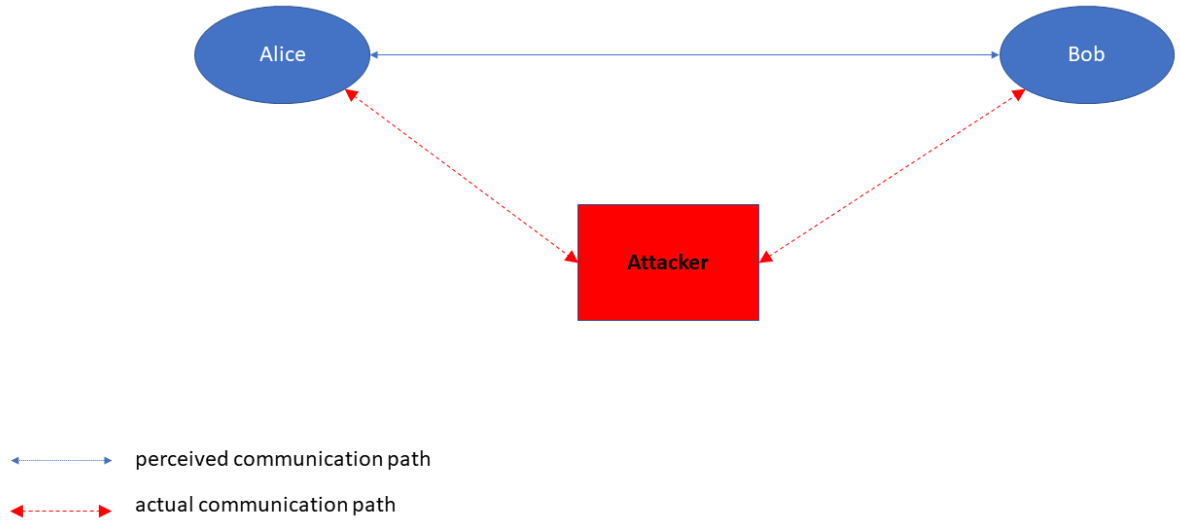


Figure 3 - The MITM attack

There are various ways in which authentication can be accomplished in computer systems. Websites typically use *digital certificates*. Possession of a shared secret, which only the authentic device and your device know, is another approach.

Integrity

How do I know the data I received is the data that was sent?

There are a number of ways in which transmitted data could change as it is being communicated from one device to another. It could become corrupted as a consequence of interference. Or it could be deliberately tampered with, changing its value in some way that is advantageous to an attacker.

Ensuring changes to transmitted data cannot occur without being detected is what *integrity* is all about. It's about ensuring data can be trusted and about making sure data corruption is detected and dealt with.

Authorization

Am I allowed to do that?

This issue is all about having permission to do something. You encounter this concept in all walks of life, but, in the context of Bluetooth communication between devices, authorization is concerned with controlling whether or not one device is permitted to read or write a data item belonging to another device through an interaction that takes place over a connection.

Privacy

Can my identity be determined by others? Can my movements be tracked by others?

Communicating devices (whether they use wired or wireless communication, Bluetooth, Wi-Fi, or something else) often include data in their communication that is designed to identify the devices involved. Source and destination addresses are extremely common in communications protocols and identify the transmitting device and the address(es) of the intended recipient device(s).

But what if someone eavesdropping on communications used that identifying data to build a picture of the typical movements of devices? What if they were able to associate a device address with personal identifying information, such as the name of the person that uses that device?

In the context of communications systems and security, *privacy* is concerned with ensuring that devices and their users cannot be tracked by unauthorized third-parties.

3.2 Cryptology, Cryptography, and Security Fundamentals

Cryptology is "the mathematical science that deals with cryptanalysis and cryptography" – see https://csrc.nist.gov/glossary/term/cryptology.

Cryptography is a subset of cryptology and various definitions exist, including this one: "Art or science concerning the principles, means, and methods for rendering plain information unintelligible and for restoring encrypted information to intelligible form" from https://csrc.nist.gov/glossary/term/cryptography.

Cryptology is at the heart of most computer-based security systems and is central to most Bluetooth security features. To understand Bluetooth security, an understanding of some fundamental ideas and terminology from the world of cryptology is required.

Alice and Bob

In the literature of cryptology, when describing an interaction between two parties, it is typical to refer to one of them as Alice and the other as Bob. Sometimes, Charlie also gets a mention. Charlie is typically an attacker.

Where useful, the same nomenclature is used here.

Keys

Cryptographic algorithms often involve *keys*. Keys are just numbers, but typically, they are large numbers. They act as an input parameter to an algorithm and affect its output. Keys allow secret information to be locked and unlocked.

Symmetric vs Asymmetric

Broadly speaking, there are two main families of cryptographic algorithms. The first is termed symmetric and the second asymmetric.

Symmetric algorithms use a single key that both the sender and recipient of information must possess. The key is sometimes called a *shared secret*.

Asymmetric algorithms involve each participant in an exchange of data, possessing a pair of mathematically related keys. One of the two keys in the key pair is called the *public key* and the other is called the *private key*. The nature of the mathematical relationship between public and private keys is such that it is not computationally feasible to derive the value of a private key solely from the public key. Public keys are distributed to others by the key pair's owner, but the corresponding private key is not. Instead, it must be kept in a secure location that only the key pair owner can access.

There are several asymmetric cryptosystems in popular use, including RSA and elliptic-curve cryptography. Bluetooth LE uses the elliptic-curve cryptosystem.

Asymmetric cryptography is also known as *public key cryptography*.

Key Distribution

Using a symmetric algorithm requires both communicating parties to be in possession of the same secret key with which to encrypt and decrypt information. Bringing this about in such a way that an unauthorised third party cannot come into possession of the secret key is a challenge. Procedures for the secure distribution of a key from one party to another (or material from which the key can be derived) fall under the heading of *Key Distribution*. See https://csrc.nist.gov/glossary/term/key-distribution.

Asymmetric cryptography takes a different approach. A public key may be freely distributed without the need to prevent third parties from acquiring the key since it can be used for encrypting information but not decrypting it. Private keys, on the other hand, are not distributed.

Encryption

Encryption is one application of cryptography and, perhaps, the most well-known. Encryption encodes information so that an unauthorized third party coming into possession of the encoded information cannot decode it and access the original information. Encryption addresses the need for *confidentiality* by ensuring sensitive information can be transmitted in the presence of eavesdroppers without them being able to access it while allowing the intended recipient of the data to decode it.

Many different encryption algorithms exist. In general, unencrypted data provided as an input to a cryptography algorithm is called *plaintext,* and the encrypted output is called *ciphertext*. A *key* is also provided as an input to an encryption algorithm.

Decryption is the name given to the act of reversing encryption so that a given *ciphertext* is returned to its original *plaintext* form. When using a symmetric encryption algorithm, the same key is used for both encrypting and decrypting data. When using an asymmetric approach, one key is used to encrypt the data and the other key from the public-private key pair is used to decrypt it.

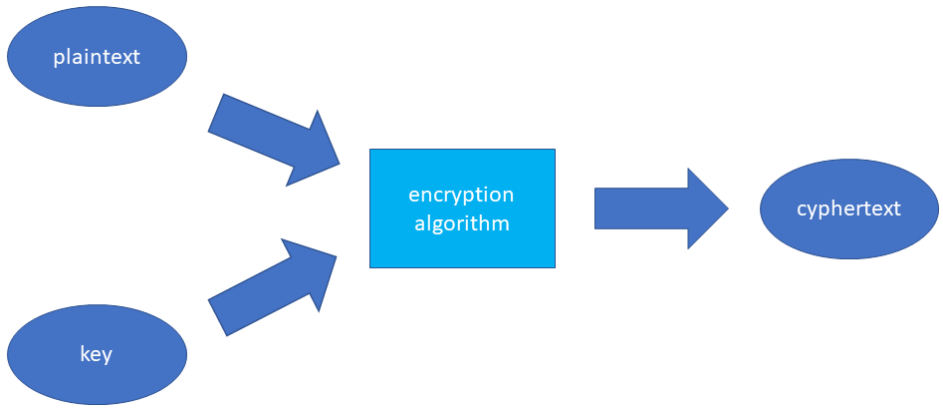


Figure 4 - Encryption

Hash Functions

Hash functions take a sequence of bytes of arbitrary length as input and output a fixed length pattern of bits called a *hash*. Hash functions are sometimes said to perform *one-way encryption* since the process of producing a hash from a given input cannot easily be reversed. Good hash functions produce unpredictable changes in the output hash value for even small changes to the

input. Hash values are likely to be unique if the hash is sufficiently large (128 bits is common), given the set of possible inputs. Consequently, sometimes the term *fingerprint* is used instead of *hash*. You may encounter the term *message digest*, which is another name for a hash function.

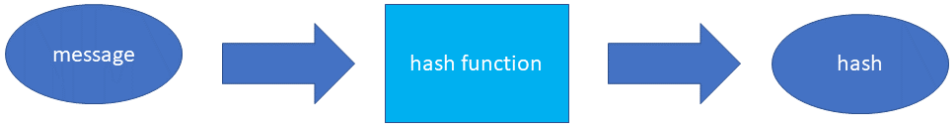


Figure 5 - Hash functions

Message Authentication Codes

Message Authentication Codes, or MACs, are used to address issues of authentication and integrity.

Note that in computing, MAC can also mean Media Access Control. Consequently, to avoid confusion, Message Authentication Codes are often called Message Integrity Codes (MICs) instead. This is the case with the Bluetooth Core Specification where the term MIC is preferred.

MACs are generated from an input message and a symmetric key, as shown in Figure 7. MACs allow changes in messages to be detected.

Imagine that Alice has a message she wants to send to Bob and that she wants to ensure the message cannot be changed while in transit without this being detected by Bob. Figure 6 illustrates the steps involved in using a MAC for this purpose.

```
1. Alice computes the MAC of her message using a key that was previously shared with Bob.

2. Alice sends the message and its MAC to Bob.

3. Bob calculates a MAC for the received message using the key pre-shared with Alice.

4. Bob compares the MAC received from Alice with the MAC he just calculated. If the two
values are the same, Bob can conclude that the message did not change in transit. If they
are not the same, Bob concludes something changed or that the MAC was not created with the
key shared with Alice. Bob cannot tell whether the change was accidental or deliberate, or
which part of the message changed, only that there has been a change or the wrong key was
used. In either case, the message is not to be trusted and is typically discarded.
```

Figure 6 - Using Message Authentication Codes

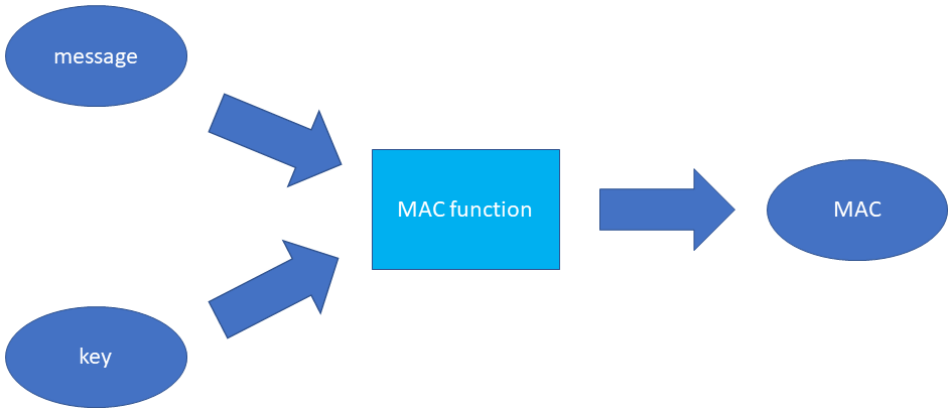


Figure 7 - MAC functions

An HMAC is a MAC function based upon a hash function. HMACs produce MAC values that are relatively small and have a fixed length series of bytes.

AES-CMAC is an example of a message authentication code generating algorithm. It is used in *LE Secure Connections*, which will be examined later in this document.

Signatures

Digital signatures are similar to MACs except that they use an asymmetric approach with public and private key pairs rather than a symmetric shared secret as is the case with MACs. Generally, if Alice wants to be able to send signed messages to Bob, she will proceed as follows:

```
1. Alice shares her public key with Bob.

2. Alice computes a signature for the message, using her private key.

3. Alice sends the message and its signature to Bob.

4. Bob verifies the signature using a suitable verification algorithm that involves the
received message and Alice's public key.

If verification is successful, Bob can conclude that the message has not changed and that
the sender was able to use Alice's private key to sign the message.
```

Information Leakage

NIST defines information leakage as *the intentional or unintentional release of information to an untrusted environment*. See https://csrc.nist.gov/glossary/term/Information-Leakage.

Information leakage can take many forms.

As an example, consider the case where a user must log in to a system using a username and password. Error messages, returned when an entered username and password are deemed invalid by the system, could leak useful information to an attacker if not chosen carefully. *Invalid user name* tells us that the user name has not been found in the database. But *invalid password*, displayed when a different user name has been entered, may tell us that this time the entered user name was

a valid one. An attacker can use this information to significantly reduce the combinations of user name and password being tried in a type of attack known as a *brute-force attack*.

Furthermore, consider the fact that encrypting a transmitted message will safeguard the confidentiality of the content of the message. But the act of transmitting the message still leaks information. It can be readily deduced that the two parties are communicating, possibly valuable information in and of itself. And the date and times at which communication between the two parties takes place may also be valuable, inadvertently leaked information.

Cipher

Series of transformations that converts plaintext to ciphertext using the Cipher Key.

https://csrc.nist.gov/glossary/term/cipher

Block Cipher

A symmetric-key cryptographic algorithm that transforms one block of information at a time using a cryptographic key. For a block cipher algorithm, the length of the input block is the same as the length of the output block.

https://csrc.nist.gov/glossary/term/block-cipher

Block ciphers are not usually used directly. Instead, they act as the building blocks for more useful algorithms called *block cipher modes*.

Stream Cipher

A cipher that operates on a continuous stream of symbols.

https://en.wikipedia.org/wiki/Stream_cipher

Block Cipher Mode

An algorithm for the cryptographic transformation of data that is based on a block cipher. See https://csrc.nist.gov/glossary/term/block-cipher-mode-of-operation

Block ciphers can handle blocks that are not all the same size and sometimes combine encryption with authentication.

Initialization Vector (IV)

Some block cipher modes will produce identical blocks of ciphertext for the same plaintext block input. This is highly undesirable as it constitutes a form of *information leakage*. Imagine an attacker has obtained a collection of unencrypted emails and notes that they always end with the same, standard email signature. Armed with this information and an awareness that the block cipher used to encrypt emails from the same organisation exhibits this unfortunate trait, an attack on the encrypted emails becomes easier.

More secure cipher block modes use schemes to avoid this, such as XORing the current plaintext block with the previous block in the sequence. This leaves an issue regarding the first plaintext block, however. There is no previous block in the sequence. Therefore, a special value that acts as the previous block for the purposes of initializing the block cipher mode is used. This value is called the Initialization Vector, often abbreviated to IV.

For a formal definition of Initialization Vector see https://csrc.nist.gov/glossary/term/Initialization-Vector.

Session Key Diversifier

A session key diversifier (SKD) is used as a parameter to a special function, known as a *key derivation function,* to generate a new key for use in each distinct session. The diversified key is a new key derived from the original secret key plus some random data (known as the *diversifier*), which is different each time diversification is required. Both parties involved in establishing the encrypted session know the SKD value to use through some mechanism.

Key diversification ensures that the key used for encryption during each session is different. This, in turn, makes sure that the same plain text does not produce the same ciphertext each time it is encrypted across different sessions, as this would constitute *information leakage*.

Nonce

A random number that should be used no more than once and which is incorporated in some cryptographic procedures to protect against *replay attacks*.

See https://csrc.nist.gov/glossary/term/nonce.

Random Numbers

The U.S. National Institute of Standards and Technology (NIST) defines a random number as "…*a value in a set that has an equal probability of being selected from the total population of possibilities and, hence, is unpredictable. A random number is an instance of an unbiased random variable, that is, the output produced by a uniformly distributed random process*".

See https://csrc.nist.gov/glossary/term/Random-Number.

Random numbers are a critical ingredient in many cryptographic algorithms and protocols. Not all methods of generating random numbers are equal, however. Some generate sequences of numbers that are more random than others. *Less random* means *more predictable*, and the ability to predict random numbers used in cryptography is bad since it can significantly weaken the security of the algorithm.

Worse still, some random number generators generate completely predictable random numbers, an apparent contradiction in terms. Such algorithms are called pseudorandom number generators (PRNGs). PRNGs work by generating a very long sequence of numbers which is dependent upon an initial value selected by the user of the PRNG called a *seed*. Usually, the PRNG will generate a different sequence of numbers for each possible seed value. Using the PRNG then involves moving along the sequence each time a new random number is needed. At first sight, the series of numbers returned by the PRNG appears random. But another application choosing the same seed will be provided with exactly the same sequence of numbers. A relatively straightforward attack on a cryptographic system that is suspected of using a PRNG is to guess the seed value and thus the random numbers.

I once worked on project whose goal was to evaluate the security of a new internet banking system in the 1990s. One of the issues uncovered, without access to the system's source code, was that a PRNG, seeded with the current time since 1st January 1970, expressed in milliseconds was being used to generate random numbers for an encryption algorithm. The current time is quite

predictable, and so this was a major weakness in the encryption system used by this internet banking system.

PRNGs have their uses, but developers need to use them with care and only where they are deemed fit for purpose.

Product manufacturers need to assure themselves that the random numbers they rely on for critical applications, such as cryptographic algorithms, possess sufficient randomness. If using APIs, such as those provided by module manufacturers, obtain information about how those APIs work and how random the generated random numbers really are. Similarly, if assessing cryptography APIs, one of the questions to ask is about how random the numbers it uses are.

One approach to generating highly random numbers is to use a hardware random-number generator (HRNG). HRNGs exploit physical phenomena that exhibit statistically random behaviors, such as the signals associated with thermal noise.

Entropy

Random numbers are important in many cryptographic algorithms. But some random numbers are more random than others and knowing how random a number is, can be important. The measure of randomness used in cryptography is known as *entropy*.

If, for example, you have a 128-bit number generated by a process for which any of the possible 2^{128} values is equally likely to be produced, then your 128-bit number is truly random, and you can say that you have 128 bits of entropy. If, however, you know that the first 64 bits are always all zero, then you would say that your 128-bit number only has 64 bits of entropy.

Generally speaking, more entropy in random numbers enables better security in an algorithm, whereas low levels of entropy will mean that random numbers are easier to guess and thus security algorithms will be weaker.

NIST defines entropy as *a measure of the amount of uncertainty that an Attacker faces to determine the value of a secret*. See https://csrc.nist.gov/glossary/term/Entropy.

Diffie-Hellman

Secret keys, shared by two parties who want to communicate securely, work well. But there's a problem. How do Alice and Bob agree the value of the key that they will use to securely communicate with each other without an eavesdropper stealing the secret key while they discuss it?

The answer is that Alice and Bob will use a *key agreement protocol* which allows them to jointly derive a secret key without actually exchanging the key, and without risking the key or the values necessary for its calculation being intercepted and stolen.

A commonly used key agreement protocol is called the Diffie-Hellman (DH) key agreement protocol.

DH has no protection against MITM attacks (see 3.1 Authentication), and so it is common to see it used in conjunction with some form of authentication.

There is a specific form of the DH key exchange protocol that makes use of *Elliptic Curve Cryptography*.

NIST defines Diffie-Hellman as *a method used to securely exchange or establish secret keys across an insecure network*. See https://csrc.nist.gov/glossary/term/Diffie_Hellman.

Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is an approach to public key cryptography based upon the mathematical properties of elliptic curves. It requires relatively small key sizes and is well suited for use by devices whose computing power is limited.

NIST defines ECC as *the public-key cryptographic methods using operations in an elliptic curve group*. See https://csrc.nist.gov/glossary/term/Elliptic-Curve-Cryptography.

Replay Attacks

A replay attack involves an attacker capturing packets communicated between two devices and then, at a later time, retransmitting or *replaying* them to fool one of the original devices into thinking it is taking part in a legitimate exchange with a trusted device.

Replay attacks are often addressed in communications protocols by including a sequence number in packets, which is incremented each time a distinct packet is transmitted. Receiving devices must reject any packet with a sequence number that is not higher than the last one received.

NIST defines a replay attack as *an attack in which the Attacker is able to replay previously captured messages (between a legitimate Claimant and a Verifier) to masquerade as that Claimant to the Verifier or vice versa*. See https://csrc.nist.gov/glossary/term/replay-attacks

4. Security in Bluetooth® Low Energy

4.1 Overview

We'll start by positioning a few important concepts and terms and then present an overview of the security features of Bluetooth LE and how they relate to the security concepts which were covered earlier.

Referring to Devices

The Bluetooth Core Specification uses different terms to refer to the two devices involved in communication. To the uninitiated, this can be confusing at first. The reason different terms are used for the same device has to do with context. For security-related interactions, such as pairing, the terms i*nitiator and responder* are often used. Usually the device which advertises and accepts connections is referred to as the Peripheral device whilst the device which scans and connects to the Peripheral is called the Central device. When discussing the Generic Attribute Profile (GATT) or Attribute Protocol (ATT), then the terms used will often be *client* and *server* instead.

Initiator and Responder

The term i*nitiator* is used in the Bluetooth Core Specification to refer to the device that is initiating a connection with another device. *Responder* refers to the other device, with which the connection will be established. In describing security procedures, the i*nitiator* is often used to refer to the device that is initiating an encryption procedure.

Central and Peripheral

The Central device is a device which is in an active connection with another device known as the Peripheral. The central device initiated the connection to the Peripheral. The Central controls timing and general coordination of data communication over the link.

GAP Roles

Central and Peripheral are two of the four GAP roles, the others being *Broadcaster* and *Observer*. *Peripheral* devices may be connected to. *Central* devices scan for advertising peripherals so that they may connect to them. *Broadcaster* and *Observer* are roles relating to connectionless communication.

Client and Server

GATT allows a hierarchical database of device attributes to be defined. The Attribute Protocol allows data in this database to be accessed and communicated in various ways. The device that hosts the attribute database is called an *attribute server* or just *server,* whereas the other device, accessing the database using ATT, is called an *attribute client* or simply *client*.

Mandatory Security Features

In the Bluetooth Core Specification, none of the security features of Bluetooth LE are mandatory. Some of the Bluetooth profile specifications do mandate minimum security requirements, however. Product and custom profile designers must assess their security requirements and include those Bluetooth security features needed to address those requirements.

Developers and Bluetooth Security

Developers who develop embedded code for Bluetooth peripheral devices have far more options (and responsibility) for choosing and using the appropriate Bluetooth security features than

application developers, such as the developers of smartphone applications, which work over Bluetooth with a connected peripheral device.

Embedded software engineers work with an SDK relating to the Bluetooth module selected for use within their product. The SDK should provide APIs and configuration options that make Bluetooth security features available to the developer, and it is up to the developer to include those features by coding for them accordingly.

More often than not, embedded software engineers develop *GAP peripheral* devices that act as *ATT servers*, containing an attribute table. Security of the peripheral device is addressed by indicating appropriate parameters during pairing, setting an appropriate *security level* and defining the right *attribute permissions* in the attribute table. These concepts will be explained later.

Application developers usually develop *GAP central* devices that act as *ATT clients*. They often find they have few or no opportunities to influence security as it relates to their application's communication with the remote peripheral device / ATT server. Their platform (e.g. Android) implements a set of security behaviours deemed optimal by the platform's developers but without necessarily providing APIs that allow security-related interactions to be customised by application developers.

Security Levels and Modes

The Bluetooth Core Specification introduces the terms *security level* and *security mode*. These terms refer to a combination of security attributes and requirements. They are used in defining the security requirements of "*...a device, a service or a service request*". See Volume 3 Part C section 10.2 of the Bluetooth Core Specification version 5.2.

A *security mode* may have a number of associated *security levels*. These are repeated from the specification here, for convenience:

LE security mode 1 has the following security levels:
1. No security (No authentication and no encryption)
2. Unauthenticated pairing with encryption
3. Authenticated pairing with encryption
4. Authenticated LE Secure Connections pairing with encryption using a 128-bit strength encryption key

LE security mode 2 has two security levels:
1. Unauthenticated pairing with data signing
2. Authenticated pairing with data signing

LE security mode 3 has three security levels:
1. No security (no authentication and no encryption)
2. Use of unauthenticated Broadcast_Code
3. Use of authenticated Broadcast_Code

LE Secure Connections Only mode (LE security mode 1 level 4)

These terms will become clearer as you learn more about Bluetooth LE security.

Security Manager

Bluetooth LE includes a component called the Security Manager and a protocol called the Security Manager Protocol (SMP). This protocol is involved in security procedures, such as pairing. See Bluetooth Core Specification 5.2 Vol 3 Part H.

Connections vs Connectionless Communication

Most Bluetooth LE security features apply only to devices when they are communicating using connections. Devices that communicate using GAP advertising and scanning are said to be performing connectionless communication and few security features are defined in the Bluetooth Core Specification for this mode of communication. One notable exception to this statement, concerns the use of broadcast isochronous channels. Isochronous channels were introduced in Bluetooth Core Specification version 5.2 and their security features are explained below.

Integrity Checks

All transmitted Bluetooth LE PDUs include a Cyclic Redundancy Check (CRC) that is recalculated and checked by the receiving device for the possibility of the PDU having been changed in flight. Technically, this is not a security measure since no authentication is involved and the CRC is not cryptographically protected in any way. CRCs only protect against accidental changes made to transmitted data, due to issues like background noise. They do not offer protection against changes deliberately made with malicious intent.

Pairing and Bonding

Devices may form a secure trusted relationship through a Bluetooth security procedure called *pairing*. Pairing provides each device with security resources, such as keys, which are needed to allow other security features, such as encryption, to be used. If two devices have not paired, they will be unable to encrypt the link and unable to sign data. It will also be impossible to determine the true identity of another device that is disguising itself using a privacy protection feature called *resolvable private addresses* (see Privacy and Device Tracking Protection below).

Pairing is the foundation of Bluetooth security. There are a variety of ways in which pairing may proceed, to accommodate the capabilities of the two devices and their security requirements. These will be reviewed in section 4.2.

If the security resources provided by pairing are stored for future reuse by the paired devices, they are said to be *bonded*.

Security Keys and Security Capabilities

Bluetooth LE security features – such as link encryption, privacy, and data signing – need specific security keys to have been created and shared by pairs of devices before they can be used.

How do devices acquire the keys they need to be able to use the related Bluetooth security features? There's a clue in the opening paragraph of this section in the word *pairs*. Key distribution is the primary purpose of the Bluetooth security procedure known as *pairing*.

There are three types of security key in Bluetooth LE:

LTK	Long Term Key - used in link encryption
CSRK	Connection Signature Resolving Key - used in signing data sent over an unencrypted link

IRK	Identity Resolving Key - used in the Bluetooth *privacy* feature

Each of these key types is optional.

Devices may discard the keys created during pairing after they have been used during the lifespan of the initial connection. Alternatively, they may store the keys in a database, along with some identifying data. This allows the keys associated with a device previously paired with to be restored and used again with subsequent connections. Devices that store and reuse security keys across connections are said to be *bonded*.

Encrypted Connections

A Bluetooth connection between two devices may be *encrypted* so that all data passing across it in either direction retains its *confidentiality* and cannot be accessed by eavesdroppers. Bluetooth LE uses an authenticating encryption algorithm called AES-CCM and, therefore, the authenticity of data exchanged using an encrypted connection is also assured.

Device Authentication

When devices are *paired*, authentication may be involved in the procedure. This means that an aspect of the pairing procedure will be concerned with verifying that the device the user thinks they are pairing with really is that device and not an imposter (a form of MITM attack, as described in section 3.1).

Authentication of Data

Packets exchanged over an encrypted link have their authenticity safeguarded due to the use of the AES-CCM authenticating encryption algorithm by Bluetooth.

Data signing may also be used with individual GATT write procedures via the ATT Signed Write command. This may be used to allow the receiving device to verify the authenticity of the attribute value in the ATT command but not of the packet as a whole.

Privacy and Device Tracking Protection

Devices may use a special form of address called a *private address*. An address of this type changes periodically and, as such, disguises the true identity of the device that is transmitting it. This prevents that device from being tracked and helps to protects the *privacy* of the device user.

Attribute Permissions

The Generic Attribute Profile (GATT) allows devices to offer an interface to selected internal data and capabilities expressed in terms of a hierarchical structure that consists of *services, characteristics,* and *descriptors*. Services, characteristics, and descriptors are different types of a*ttribute* and the collection of attributes a device has is organized in an internal table called the *attribute table*.

Readers unfamiliar with GATT should read the Bluetooth LE basic Theory document, part of the Bluetooth LE Developer Study Guide, for an introduction to the key terms and concepts.

All attributes include a series of permissions governing the way they can be accessed and the rules that apply to access, including the degree to which the other device must be trusted. A device

authenticated during pairing is deemed more trustworthy than one that was not. So, for example, an attribute permission may stipulate that only devices that were authenticated during pairing may read the attribute or write to it.

The Accept List

The Bluetooth LE link layer in the controller possesses a feature known as the *accept list*. The *accept list* is a list of device addresses and their type. Its purpose is to allow the link layer to perform device filtering of various types, referred to as *filter policies*. See Bluetooth Core Specification 5.2 Vol 6, Part B 4.3.1.

The accept list is primarily aimed at reducing the amount of work performed by the stack in handling packets from devices of no interest and, therefore, reducing power consumption. It can also be used to thwart unsophisticated denial of service attacks.

GAP Peripherals can typically only accommodate one connection at a time. Therefore, an attacker could prevent another GAP Central device from connecting simply by connecting to the Peripheral device first. To address this issue, using the *advertising filter policy*, the accept list can prevent devices that have not been paired from establishing a connection. All that is required to achieve this is to add an entry to the accept list for each paired device and then enable the advertising filter policy. A connection request from any device not included in the accept list will be ignored.

Isochronous Channels

Isochronous channels are used for the communication of data whose processing must be time-synchronized by the devices receiving it (e.g. LE Audio). When used with connected devices, isochronous channels may use the same security features available in any other connected device scenario.

When isochronous channels are used in connectionless mode, devices are said to form a *broadcast isochronous group (BIG)*. A BIG may use LE security mode 3, level 1, 2 or 3. Level 1 offers no security. Level 2 provides unauthenticated encryption and level 3 provides authenticated encryption of all broadcast data to devices in the BIG.

Encrypting data in a BIG requires the use of a *broadcast code*. This 16-octet value is obtained in a way which is defined by a profile specification and then passed from host to controller. An encryption key is generated from the *broadcast code* and used to encrypt all data broadcast to devices in the BIG.

Section 4.4.6.10 of Bluetooth core specification 5.2 defines how all non-empty isochronous broadcast PDUs in a BIG are encrypted.

4.2 Examination

Armed with an awareness of key security concepts and features of Bluetooth LE, you will now examine those features in more detail. References to Bluetooth Core Specification version 5.2 will be given from time to time and functions defined in Volume 3, Part H section 2.2 “Cryptographic Toolbox” will be mentioned.

Pairing

Pairing is the foundation of Bluetooth LE security, so we’ll next spend some time exploring the subject in detail. Most of this section is covered in Bluetooth Core Specification version 5.2, volume 3, part H.

There are a number of different ways in which pairing can proceed. This is apparent both in terms of variations in the user experience and in the messages exchanged between the two devices.

Important: You should assess security requirements for the pairing process itself, separately from the security requirements for use of the device after pairing has been performed. The alternative ways in which pairing can proceed offer v*ery* different degrees of security.

The key to understanding Bluetooth pairing is to recognize what it must achieve, summarised in three points in Figure 8.

1. Keys that enable the use of required Bluetooth security features such as encryption, data signing and privacy must be distributed to both devices for future use.

2. Distribution of up to three types of keys between the two devices must be accomplished with *protection* from passive eavesdroppers who may attempt to steal the keys being distributed.

3. It *may* be necessary to authenticate devices during pairing to protect against attacks, such as the MITM attack.

Figure 8 - Pairing goals and requirements

Alternative Pairing Methods

Devices running a Bluetooth LE stack compliant with Bluetooth Core Specification version 4.2 or later may have two distinct pairing methods available. The first is called *LE Legacy Pairing* and the second, a newer and substantially more secure method, is called *LE Secure Connections*. In both cases, there are possible variations during each of the phases of pairing according to a number of factors. This will be described shortly.

Note: Devices running a stack whose version is earlier than Bluetooth 4.2 will only be able to use the LE Legacy Pairing method.

Figure 9 shows the ways pairing can proceed at a high level. The specification breaks pairing into the three phases depicted in Figure 9. There is a decision made in phase 1, resulting in either LE Legacy Pairing or LE Secure Connections pairing being used. This is followed by phase 2, which includes an optional authentication step, the generation of a key with which to encrypt the link in all cases and then initiation of link encryption. Phase 3 is then concerned with distribution of the required keys,

which may include any one or more of the LTK, CSRK and IRK. Revisit *Security Keys and Security Capabilities* from earlier in this guide if you have forgotten these abbreviations.

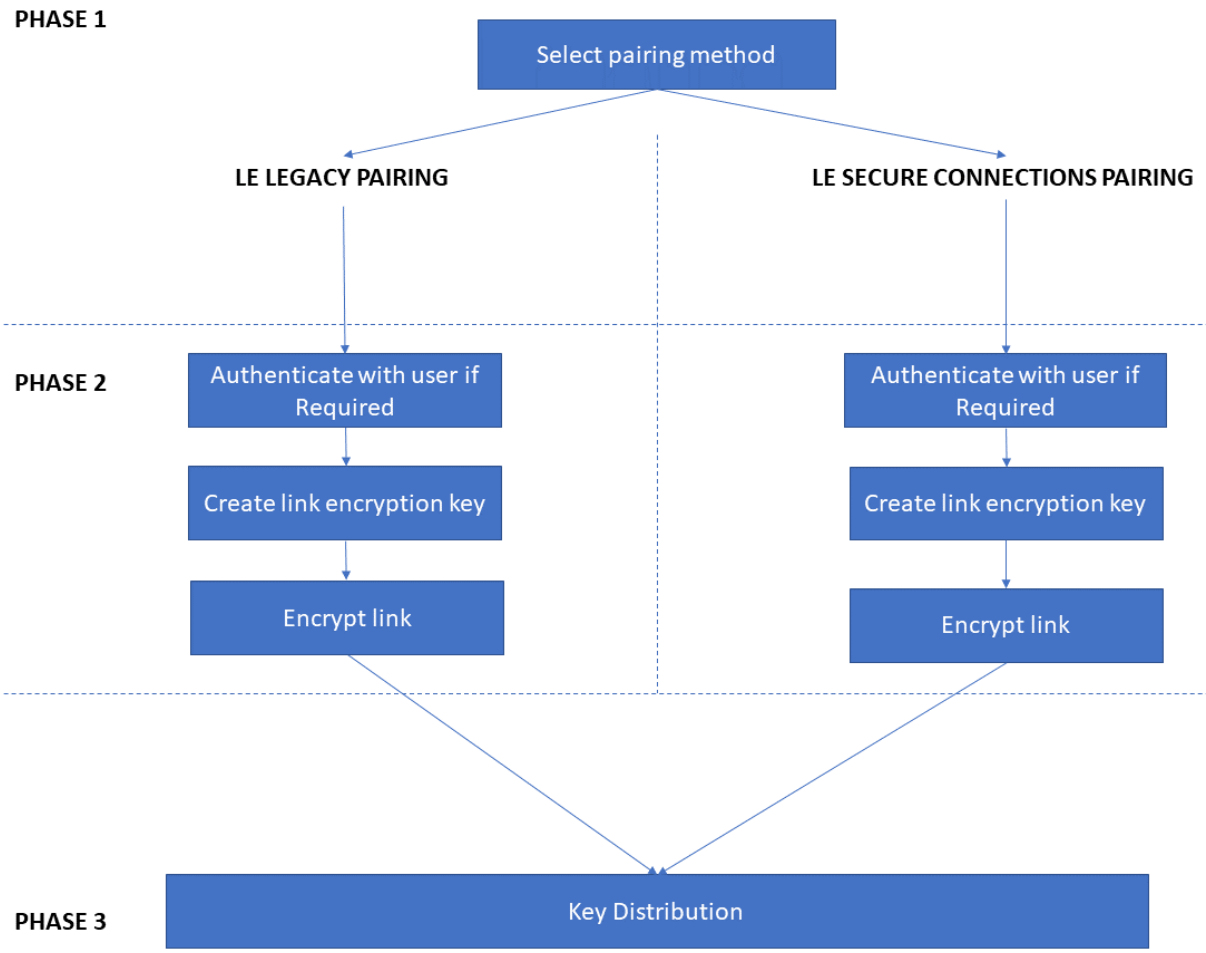


Figure 9 – High-level pairing flow

Association Models

Each pairing method offers a number of different ways in which pairing may proceed, including (where applicable) how authentication will be handled. These alternatives are called *association models*.

LE Legacy Pairing has three available association models, known as Just Works (JW), Passkey Entry (PKE), and Out of Band (OOB).

LE Secure Connections pairing offers four association models: Just Works, Numeric Comparison (NC), Passkey Entry, and OOB.

The various association models are examined later.

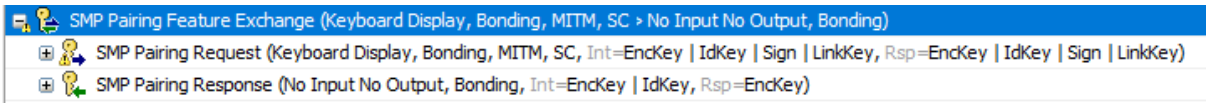
The Difference Between LE Legacy Pairing and LE Secure Connections

The two pairing methods have the same fundamental goals as listed in Figure 8. They differ from each other in two ways, however:

	LE Legacy Pairing	LE Secure Connections
Confidentiality during key distribution	Uses a simple process of exchanging secret data to derive a symmetric key with which to encrypt the link during the key distribution phase.	Uses elliptic curve public key cryptography to allow a symmetric key to be securely derived. That key is then used to encrypt the link during the key distribution phase.
Association models	Just Works, Passkey Entry, OOB	Just Works, Passkey Entry, Numeric Comparison, OOB

Pairing Feature Exchange

Pairing always starts with a step called the Pairing Feature Exchange. This involves the Initiator device sending an SMP Pairing Request PDU to the Responder and the Responder replying with an SMP Pairing Response.



This exchange provides each device with the information required to:

1. Decide whether LE Legacy Pairing or LE Secure Connections pairing should be used
2. Decide whether or not device authentication should be performed during pairing and, if so, what form that authentication step should take
3. Determine which key types should be generated and distributed
4. Determine what length a key called the Long Term Key (LTK) should be

The SMP Pairing Request and Response PDUs contain fields called *IO Capability*, SC, *MITM*, *Maximum Encryption Key Size*, *Initiator Key Distribution,* and *Responder Key Distribution*. These fields will be described shortly.

Let's review the primary details involved in the *pairing feature exchange* step and understand how they influence the way in which pairing proceeds.

IO Capability

Each device informs the other device of its IO capabilities expressed as any one of the following values:

IO Capability	Meaning
DisplayOnly	Device can display numbers or text but cannot accept input
KeyboardOnly	Device can accept text or numeric input from the user
DisplayYesNo	Device allows the user to respond with YES or NO
NoInputNoOutput	Device has no input or output capabilities a user can utilize
KeyboardDisplay	Device has both a keyboard and a display

This information affects two aspects of the pairing procedure.

Firstly, if authentication has been requested, the IO capabilities of the two devices will determine how exactly authentication will be performed. Maybe the user will see a six-digit number displayed on one device and have to enter it into the other device using its keyboard, for example.

Secondly, in the case of LE Legacy Pairing, the step that creates the encryption key for use when distributing the main keys in phase 3 requires some data with which to create a *temporary key* (TK). How that data is acquired is also determined by the IO capabilities the two devices exchange. This will be revisited shortly.

Bonding Flags

Used to indicate whether or not the device wishes to bond, i.e. to store the resultant keys for later (re)use.

SC

The SC field is part of a composite field called AuthReq. It is a one-bit flag that indicates whether or not the device supports LE Secure Connections pairing. If it is supported by a device, it *must* set this flag to 1. If both devices indicate support for Secure Connections pairing, then it *must* be used. This is the basis for the decision shown in phase 1 of Figure 9.

MITM

The MITM field is another one-bit flag within the AuthReq field. Whether LE Legacy Pairing or LE Secure Connections is used, authentication may be requested by either or both devices by setting this flag.

OOB Data Flag

Authentication can be achieved in a number of ways, and this is one of the primary variables governing how pairing proceeds. One of those ways involves the use of Out of Band (OOB) data. OOB data is data communicated to another device via a mechanism that *does not use Bluetooth*. It could, for example, involve Near Field Communications (NFC) or perhaps QR codes. Product designers wishing to use an OOB method of authentication are responsible for selecting an OOB technology that is sufficiently secure for its purpose.

The OOB data flag allows a device to indicate it is in possession of OOB data from the other device.

Maximum Encryption Key Size

Larger encryption keys provide stronger security when used with a given algorithm. However, different types of devices can support different maximum key lengths. This field allows the devices being paired to inform each other of that maximum, which will be in the range 7 to 16 octets (56 to 128 bits). Both devices need to use the same key length, and, therefore, the smaller of the two values is used for key generation.

Initiator Key Distribution and Responder Key Distribution

A device uses these fields to indicate the types of key it would like to provide and the types of key it is requesting from the other device. The available types of key are LTK, CSRK, and/or IRK.

Selecting the Pairing Method

If Secure Connections pairing can be used because both devices support it, then it must be used.

If this is not the case, then the precise path followed depends on a combination of IO Capability values, the OOB data flag, and the MITM flag from each device. The Bluetooth Core Specification defines the rules for selecting that path in matrix form in Vol 3 Part H section 2.3.5.1. Those details are not repeated here. Now would be a good time to download and open the Bluetooth Core Specification if you have not already done so.

LE Legacy Pairing

An Overview of LE Legacy Pairing

Pairing is a three-phase process. Phases 1 and 3 work in the same way for each of the two methods, *LE Legacy Pairing* and *LE Secure Connections*. Phase 2 varies according to the method and Figure 10 shows the phase as it applies to LE Legacy Pairing.

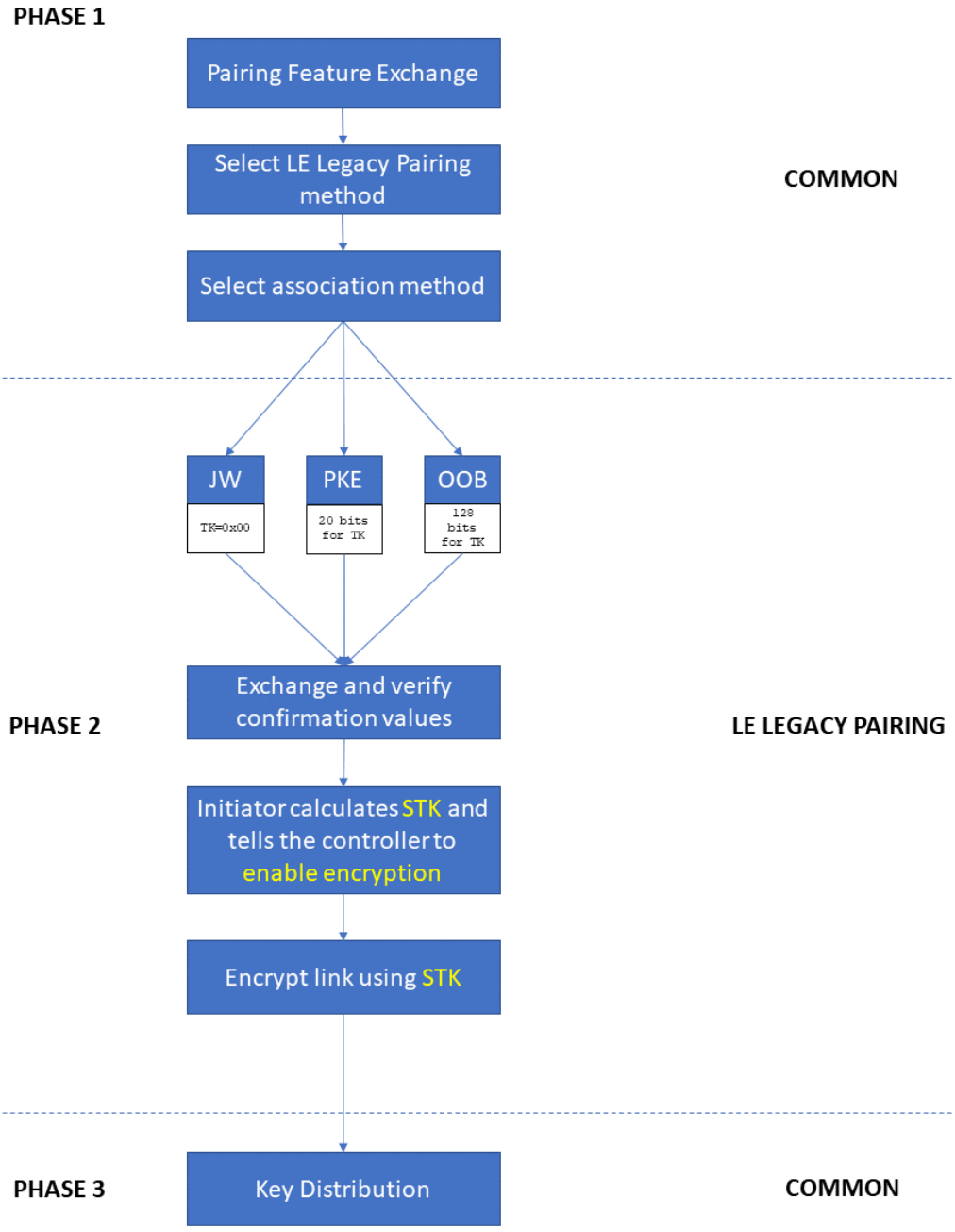


Figure 10 - LE Legacy Pairing

Phase 1

Phase 1 starts with one of the two pairing methods being selected. If one or both devices has not set the SC (Secure Connections) flag in the pairing feature exchange, then LE Legacy Pairing is used. Next, an *association model* is selected and, when LE Legacy Pairing is in use, the options are *Just Works*, *Passkey Entry,* or *Out of Band (OOB)*.

Table 1 - LE Legacy Pairing association models

Just Works	Involves no interaction with the user and it just works. Selected IF: - One or both devices have not set the SC (Secure Connections) flag - Both devices have set the MITM (Man In The Middle) flag to 0 OR - One or both devices have not set the SC (Secure Connections) flag - Both devices indicate that OOB authentication data is NOT available by both setting the OOB Data Flag to 0 - One or both devices have set the MITM flag to 1 - IO capabilities indicate that passkey entry cannot be supported (e.g. both devices have a display only and no keyboard input – see Table 2.8 in Volume 3 Part H for the full matrix of IO Capability permutations)
Passkey Entry	Requires one device to display a six-digit random number and the user to enter it into the other device. Selected IF: - One or both devices have not set the SC (Secure Connections) flag - Both devices indicate that OOB authentication data is NOT available by both setting the OOB Data Flag to 0 - One or both devices have set the MITM flag to 1 - IO capabilities indicate that passkey entry can be supported (e.g. one device has a display and the other supports keyboard input – see Table 2.8 in Volume 3 Part H for the full matrix of IO Capability permutations)
OOB	Involves passing data between the two devices in either one direction or bi-directionally using a communication channel that is not Bluetooth. Product designers are responsible for selecting the OOB channel for their product and assessing its security against requirements. It is recommended that a selected OOB channel be resistant to MITM attacks. Selected IF: - One or both devices have not set the SC (Secure Connections) flag - Both devices indicate the availability of OOB authentication data by both setting the OOB Data Flag in Pairing Request / Response PDUs to 1

Phase 2

LE Legacy Pairing encrypts the link for the purposes of securely distributing keys in phase 3, using something called the *Short Term Key (STK)*. The STK is only used for this purpose and is not retained (hence *short term*).

Deriving an STK value so that key distribution can proceed is one of two primary concerns of phase 2, as depicted in Figure 10. The other concern is authentication, whose purpose is to offer protection against MITM attacks (see Figure 3).

Phase 2 consists of the following four steps.

Step 1 - Establish a Temporary Key (TK)

The STK is derived using several values, one of which is a key called the *Temporary Key* or TK for short. We'll see how TK is used in phase 2, step 3.

TK is a 128-bit value. How it gets set depends on the selected association model and this is summarised in Table 2.

Table 2 - TK and Association Models

Association Model	TK Value	Comment
Just Works	Always zero	0 bits of entropy
Passkey Entry	The six-digit passkey entered by the user, padded with leading zeroes to give a 128-bit value.	A six-digit decimal number is represented by 20 bits. Therefore, we have 20 bits of entropy.
OOB	A full, 128-bit value that has been passed out of band between the devices.	128 bits of entropy are possible here, provided the generation of the OOB data is truly random.

Comment: TK will be used in deriving an encryption key called STK. Given the significantly different levels of entropy in TK, depending on the association model used, you can already see that the different association models result in different levels of security during the pairing process.

Step 2 - Authenticate

Authentication is achieved by each device passing the other, a special confirmation number whose calculation involves TK. In the case of Passkey Entry and OOB, TK is never passed over the air and includes data which has been shared between the two devices in a way that is immune or resistant to MITM attacks (e.g. read off the screen of one device and typed into the other by the user). So, knowledge of TK, if proven, can be taken as confirmation that the devices being paired are in direct communication with each other without a malicious third party sitting in the middle.

Here's how the process works:

To begin, each device calculates a 128-bit value known as the *confirm value*. The *confirm value* is calculated using a function called *c1* that takes various arguments, including TK. Remember that TK has not been sent over the air.

In addition to the TK value, other inputs to the function that creates the *confirm value* include fields that both devices know plus a random number that, at this stage, is only known to the device creating the confirm value. On the Central, this value is known as *LP_RAND_I.* On the Peripheral, it is known as *LP_RAND_R*.

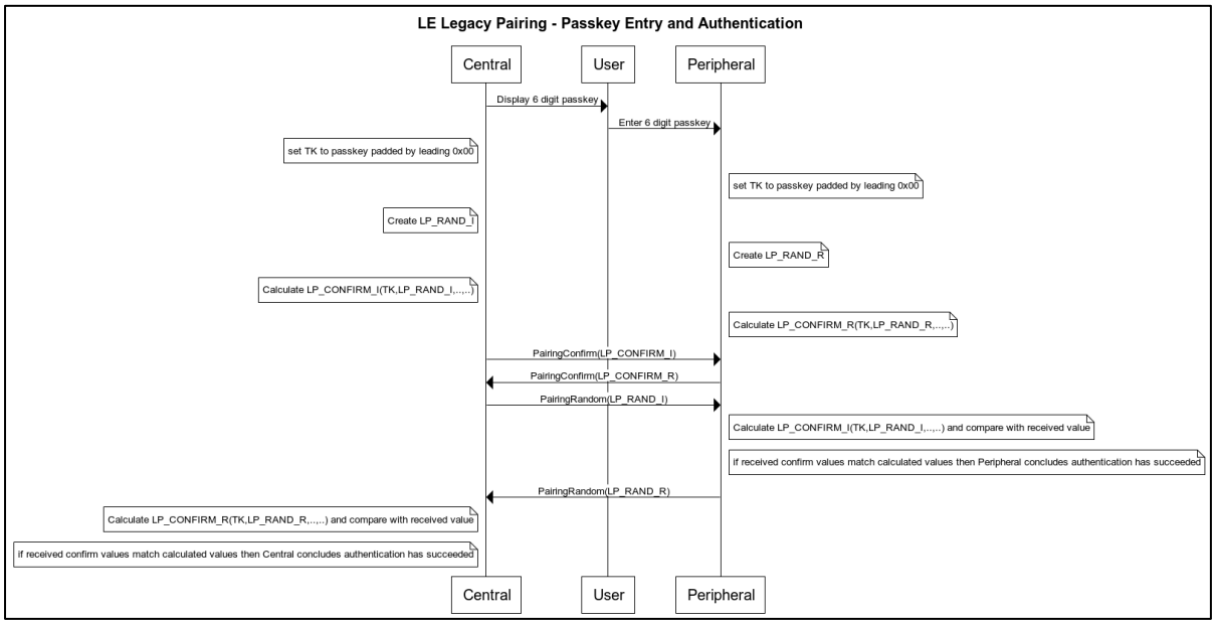
```
LP_CONFIRM_I = c1(TK, LP_RAND_I, Pairing Request command,Pairing Response command,
initiating device address type, initiating device address, responding device address type,
responding device address)

LP_CONFIRM_R = c1(TK, LP_RAND_R, Pairing Request command,Pairing Response command,
initiating device address type, initiating device address, responding device address type,
responding device address)
```

The Central device then sends its confirm value (*LP_CONFIRM_I*) to the Peripheral, and the Peripheral responds with its confirm value (*LP_CONFIRM_R*).

On receiving *LP_CONFIRM_R*, the Central shares its random number *LP_RAND_I* with the Peripheral by sending a *Pairing Random PDU*. The Peripheral now has everything it needs to calculate an *LP_CONFIRM_I* value and compare it with the *LP_CONFIRM_I* value it received from the Central in the *Pairing Confirm PDU*. If they match, then the Central has authenticated the Peripheral and so it responds with a *Pairing Random PDU* which contains LP_RAND_R. The Central calculates an LP_CONFIRM_R value and compares it with the value it previously received. If they match, then the Peripheral has authenticated to the Central.

Each device has now proved that it knows TK and, where passkey entry and OOB pairing is concerned, this means authentication has been achieved.

Just Works pairing offers **no authentication** and, hence, no protection against MITM attacks.



Step 3 - Generate the Short Term Key (STK)

Having authenticated (PKE and OOB only) and derived a value for the TK, it is now possible to create the 128-bit STK to use in encrypting the link. STK is calculated using a function which the Core Specification calls *s1*. See Vol 3, Part H 2.2.4 for details of s1.

STK = s1(TK, LP_RAND_R, LP_RAND_I)

Figure 11 - Calculating STK

As shown in Figure 11, calculating STK involves the TK plus the Central's random number *LP_RAND_I* and the Peripheral's *LP_RAND_R*. Both devices should both be in possession of all three of these arguments and, thus, be able to create the same STK value.

Step 4 - Start Encryption on the Link

In the fourth step of phase 2, the encryption is started on the link using the calculated STK to calculate a *session* key which is then used as the link encryption key. From this point on, all packets are encrypted at the Bluetooth link layer.

Phase 3

Phase 3 distributes keys of whatever type the feature exchange step in phase 1 has indicated are needed. The options are any or all of LTK, CSRK, and IRK. Key length is also determined by information exchanged in phase 1.

The link over which the keys are distributed is encrypted using a session key derived from the STK from phase 2.

Note: The link will continue to be encrypted using the STK-based session key until the next reconnection when LTK will be used instead. This could be for a lengthy period of time, depending on the use case that gave rise to pairing being triggered, with the device continuing to be used over that initial connection after pairing has completed. Think about this, given the potential weakness of STK (with an entropy of zero in the case of Just Works).

How do LTK, CSRK, and IRK get created?

Bluetooth Core Specification volume 3, Part H, Appendix B describes how LTK, CSRK, and IRK are generated and indicates that they could be generated during the device manufacturing process, on demand when pairing is in progress, or in some other way.

Key exchange may be bi-directional. Generally, the Central device starts encryption and it will be the Peripheral's LTK that is used in establishing encryption. Therefore, the Central needs to receive the Peripheral's keys. But if it is possible that in the future the two devices might reverse their roles, with the Peripheral becoming the Central and vice-versa, then key distribution will also distribute the Central's keys to the Peripheral.

A device may be paired with multiple devices, so there needs to be a way devices can identify keys and refer to them from the Security Manager Protocol. For this reason, LE Legacy Pairing defines two special security values called the Encryption Key Diversifier or *EDIV* and a Random Number or *RAND*. These values are also exchanged during phase 3. The expectation is that LTK and its associated EDIV and RAND values are stored in a security database in the device so, that when it receives a pair of EDIV and RAND values during an encryption start procedure, it is able to retrieve and use the right LTK value.

Recap – A Summary of LE Legacy Pairing

Phase 1 – Decides that LE Legacy Pairing should be used by looking at the IO capabilities of the two devices and other pairing requirements that the devices express in the pairing feature exchange procedure. The types of keys which need to be generated and distributed, so that the devices can use the security features that they want to use (i.e. encryption, privacy, data signing), is also decided during this phase.

Phase 2 – One of several approaches to acquiring material to incorporate within a temporary key (TK) is selected. The approach is called an *association model*. Optionally, the user is also involved in an interactive procedure by which the user confirms that the device they are pairing with is the device they think they are pairing with (i.e. devices are authenticated by the user). If authentication is involved, then the selected authentication procedure is also the source of the material from which to create TK. The amount of entropy in the 128-bit TK varies considerably, from 0 bits to 128 bits, depending on how TK is created (i.e. the association model used). A short-term key (STK) is now generated, using TK as one of the inputs to the STK generation function.

At this point, devices may have been authenticated and an STK has been generated.

Phase 3 – The link is encrypted with a session key derived from STK, and keys of types established in phase 1 are distributed. Sometimes this is a unidirectional process from Peripheral to Central and sometimes this is bi-directional.

Devices are now paired. If the LTK/CSRK/IRK keys are stored for later reuse, the devices are said to be bonded.

You have now reviewed LE Legacy Pairing, including its various forms, features, and how it works.

LE Secure Connections

LE Secure Connections is an alternative and significantly more secure approach to pairing Bluetooth LE devices which was introduced in version 4.2 of the Bluetooth Core Specification.

An Overview of LE Secure Connections Pairing

LE Secure Connections pairing is also a three-phase process. Phases 1 and 3 play the same part as they do when LE Legacy Pairing is in use. Phase 2 in the LE Secure Connections pairing method is substantially different to Phase 2 in LE Legacy Pairing, however. Figure 12 shows the phases as they apply to LE Secure Connections pairing.

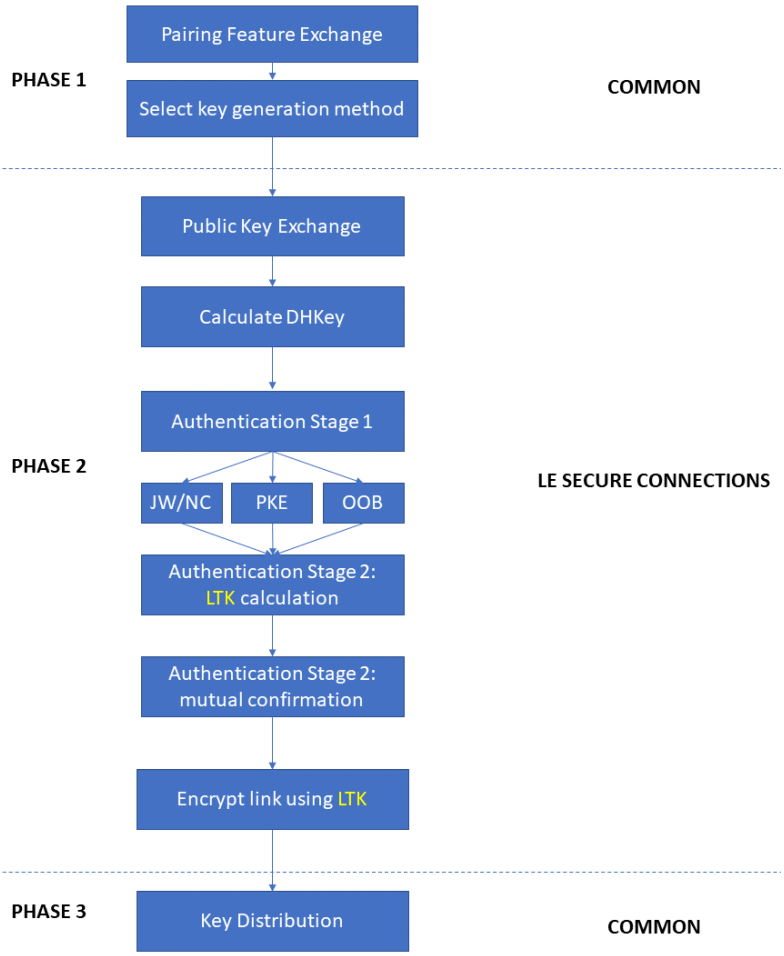


Figure 12 - LE Secure Connections pairing

As hinted at by Figure 12, LE Secure Connections pairing uses public key cryptography. More precisely, it uses elliptic-curve cryptography with the P-256 elliptic curve (see [Vol 3] Part H, Section 2.3.1).

The three phases of LE Secure Connections pairing.

Phase 1 – Pairing Feature Exchange and Pairing Method Selection

Phase 1 starts with the pairing feature exchange. If both devices set the SC (Secure Connections) flag, then LE Secure Connections pairing **must** be used. Next, an *association model* is selected using IO capabilities and other exchanged requirements (ref: Table 2.8 in Core Specification 5.2 Vol 3, Part H). When LE Secure Connections pairing is in use, the options are *Just Works*, *Numeric Comparison*, *Passkey Entry,* or *Out of Band (OOB)*. The types of keys which need to be generated and distributed,

so that the devices can use the security features that they want to use (i.e. encryption, privacy, data signing), is also decided during this phase.

Table 3 - LE Secure Connections association models

Just Works	Involves no interaction with the user and it just works. Selected IF: - Both devices have set the SC (Secure Connections) flag - Neither device has set the MITM (Man In The Middle) flag OR - Both devices indicate that OOB authentication data is NOT available by both setting the OOB Data Flag to 0 - One or both devices have set the MITM flag to 1 - IO capabilities indicate that numeric comparison and passkey entry cannot be supported (e.g. both devices have a display only and no keyboard input – see Table 2.8 in Volume 3 Part H for the full matrix of IO Capability permutations)
Numeric Comparison	The same six-digit random number is displayed to the user on both devices. The user must indicate whether or not the two numbers are identical, perhaps by pressing one button for *yes* and another for *no*. Selected IF: - Both devices have set the SC (Secure Connections) flag - One or both devices have set the MITM (Man In The Middle) flag - Both devices indicate that OOB authentication data is NOT available by both setting the OOB Data Flag to 0 - IO capabilities indicate that both devices have a display and can support a YES/NO choice from the user (e.g. with YES/NO buttons or a full keyboard – see Table 2.8 in Volume 3 Part H for the full matrix of IO Capability permutations)
Passkey Entry	Has two variants when used with LE Secure Connections pairing. Either (variant #1) the user enters the same six-digit passkey into both devices or, per LE Legacy pairing, (variant #2) one device displays a six-digit random number and the user enters it into the other device. Selected IF: - Both devices have set the SC (Secure Connections) flag - One or both devices have set the MITM (Man In The Middle) flag - Both devices indicate that OOB authentication data is NOT available by both setting the OOB Data Flag to 0 - IO capabilities indicate that both devices have a keyboard but no display (variant #1) or at least one device has a keyboard and at least one device has a display – see Table 2.8 in Volume 3 Part H for the full matrix of IO Capability permutations
OOB	Involves passing data between the two devices in either one direction or bi-directionally using a communication channel that is not Bluetooth.

	Product designers are responsible for selecting the OOB channel for their product and assessing its security against requirements. It is recommended that a selected OOB channel be resistant to MITM attacks. Selected IF: - Both devices have set the SC (Secure Connections) flag - One or both of the two devices indicate the availability of OOB authentication data by both setting the OOB Data Flag to 1

Phase 2 – Public Key Exchange, DHKey Calculation, and Authentication

Phase 2 of LE Secure Connections pairing is very different to Phase 2 in LE Legacy Pairing in a number of ways.

For example, LE Secure Connections uses Elliptic Curve Diffie-Hellman (ECDH) public key cryptography to allow the secure exchange of data; data which is then used to create a symmetric key called the Long Term Key (LTK). LTK is then used to encrypt the link over which key distribution takes place in Phase 3, and it is retained for future use.

In addition, authentication takes place in two steps. The first of which (*Authentication Stage 1)* may involve user interaction and the second (*Authentication Stage 2),* which does not, uses techniques that rely on proving possession of the correct private keys instead. In contrast to LE Legacy pairing, data input by the user during authentication stage 1 is not used as a source of data for deriving cryptographic keys, with the varying levels of entropy, and thus security, that this approach entails. Data input by the user is only used in authentication and has no further use beyond that.

Each device must possess or have the ability to generate an ECDH public-private key pair. The specification recommends that a device change its private key after every pairing, whether it was successful or not, or, alternatively, change its private key whenever S + 3F > 8 where S is the number of successful pairings and F the number of failed attempts since the key was last changed (ref Bluetooth Core Specification 5.2 Vol 3 Part H 2.3.6).

LE Secure Connections pairing is significantly more secure than LE Legacy Pairing (See *An Appraisal of Bluetooth Pairing and Security* elsewhere in this document), and, therefore, LE Secure Connections is the preferred way to go about pairing devices wherever possible. It should be noted that this is more than just a recommendation. If two devices have the ability to use LE Secure Connections pairing, then it is mandatory that they do so (ref Bluetooth Core Specification 5.2 Vol 3 Part H 3.5.1 in the section on the SC field).

LE Secure Connections pairing phase 2 breaks down into several steps.

2.1 Public Key Exchange

The device initiating pairing (the *Initiator*) sends its public key to the other device (the *Responder*), which replies with its public key. The SMP Pairing Public Key PDU is used in each case. Public keys are validated on receipt to check they are on the correct elliptic curve. Note that LE Secure Connections uses the P-256 curve only.

2.2 Calculate DHKey

A key called the Diffie-Hellman key (DHKey) is now calculated by each device. The two devices, Alice and Bob, calculate DHKey as shown here:

```
Alice: DHKey = P256(SKa, PKb)
Bob  : DHKey = P256(SKb, PKa)
```

The private keys of the two devices, A and B, are denoted *SKa* and *SKb* where SK stands for Secret Key. PK stands for Public Key. The function P256 is defined in section 7.6, Volume 2 Part H of Bluetooth Core Specification 5.2.

DHKey is a symmetric, shared secret, i.e. both devices possess the same DHKey value after it has been calculated.

Note, in calculating DHKey, each device uses its own private key (SK) and the public key of the other device received during the public key exchange step. Remember that private keys are never disclosed to other devices. You will see how DHKey is used later in authentication stage 2.

2.3 Authentication Stage 1

This stage proceeds according to the association model selected with reference to the details exchanged in the pairing Feature Exchange in phase 1. Available options are Just Works, Numeric Comparison, Passkey Entry, and OOB. See Table 3 above for a reminder about these options.

This stage allows the user to verify that the device they are pairing with is the one they think they are pairing with, unless Just Works has been selected, which is not able to provide the user with this opportunity.

2.3.1 LE Secure Connections with Just Works and Numeric Comparison Authentication

Now, it is time to review in more detail what happens when either Just Works or Numeric Comparison authentication is in use. Except for one step, these two association models are handled in the same way.

See Bluetooth Core Specification 5.2 section 2.3.5.6.2 Authentication stage 1 – Just Works or Numeric Comparison and Figure 2.3 for a diagram depicting the sequence of events.

To begin with, device B generates a pseudo-random, 128-bit number or *nonce* called *Nb* and uses it along with both devices' public keys as arguments to a function called *f4* to calculate a *confirm value* known as *Cb*.

```
Cb=f4(PKb, PKa, Nb, 0)
```

About function f4:

f4 is the *confirm value generation function*. It uses the AES-CMAC message authentication code algorithm with the nonce (Nb in the above example) as its 128-bit key. The message for which the MAC is generated is formed by concatenating the other function arguments.

Cb is then sent to device A. Device A can do nothing with the confirm value at this stage since it does not know the value of *Nb*.

Device A now sends its own pseudo-random number, *Na* to device B, and device B replies with its random number, *Nb*. Now, device A possesses all of the parameters to the f4 function, so it recalculates the confirm value using the public keys acquired during public key exchange and the other device's random number, *Nb* which it just received. It compares its calculated confirm value with the value of *Cb* received from device B. If they are not the same, pairing is aborted.

If Just Works is being used, Authentication Stage 1 has completed. If Numeric Comparison is being used, there is one more step to perform.

Each device uses the two public keys, *PKa* and *PKb,* and the two pseudo-random numbers *Na* and *Nb* to calculate a six-digit number using a function called *g2*. Each device then displays the number it has calculated. The user is invited to confirm that both devices are displaying the same six digits, perhaps by pressing a particular button. By indicating that the two numbers are the same, the user confirms that the two devices involved in the communication are indeed the devices the user is attempting to pair. In other words, the user participates in the authentication of the devices being paired. Having authenticated the devices, we proceed to the Authentication Stage 2.

If, on the other hand, the user indicates the two numbers are not the same, authentication stage 1 has failed and the process is aborted.

2.3.2 LE Secure Connections with Passkey Entry

Ref Bluetooth Core Specification 5.2 section 2.3.5.6.3 Authentication stage 1 – Passkey Entry

Passkey Entry, when used with LE Secure Connections, may involve a different user experience (UX) compared with Passkey Entry when used with LE Legacy Pairing. There are two variants available. In the first variant, a passkey is generated and displayed on one device, and the user must enter it into the other device. This is the same UX as when Passkey Entry is used with LE Legacy Pairing. Alternatively, if neither device has a display but both have suitable keyboards, the user may be required to enter the same passkey into both devices.

After the user has entered the passkey or passkeys, devices calculate, exchange, and check confirm values called *Ca* and *Cb*. But whereas confirmation happens in a single pass with Just Works/Numeric Comparison, with Passkey Entry, the confirmation process is iterative and incremental.

During each iteration, each device generates a 128-bit random number called *Na* or *Nb,* and one bit of the passkey value is incorporated in the calculation of the confirm values at a time. Inputs to the confirm value function, f4, include both public keys, the random number for this iteration, and a value derived from this iteration's passkey bit value.

Every iteration confirm values are exchanged and checked. A six-digit passkey requires 20 bits, so the process of calculating, exchanging, and checking the full confirm value takes 20 iterations with a new bit of the passkey incorporated or *disclosed* at each iteration. This approach, known as *Gradual Disclosure,* has the benefit that MITM attacks are more difficult in practical terms with most attacks failing early without the attacker having *seen* the entire and final confirm value.

2.3.3 LE Secure Connections with OOB

Ref Bluetooth *Core Specification 5.2 section 2.3.5.6.4 Authentication stage 1 – Out of Band*

With the OOB association model, part of the overall process, is conducted using a data-exchange mechanism that is not Bluetooth. This is the Out of Band part. The remainder of the procedure is conducted using Bluetooth and Security Manager Protocol PDUs.

2.3.3.1 The Out of Band Part

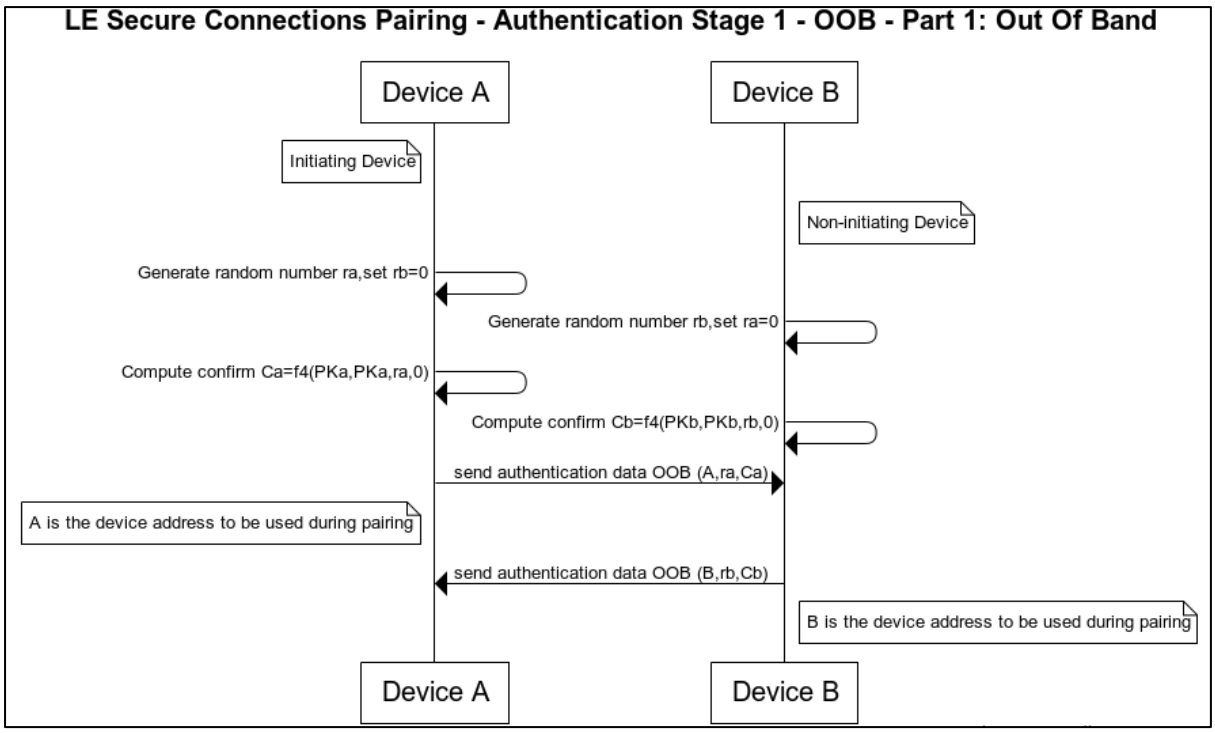


Figure 13 - The Out Of Band part of LE Secure Connections Pairing OOB

OOB with LE Legacy Pairing requires both devices to provide authentication data OOB. OOB when used with LE Secure Connections allows either both devices or only one of the devices to provide authentication data OOB.

The following description applies to the case where both devices have OOB authentication data available. When only one device has OOB authentication data available, then the communication of authentication data will be from one device to the other only and not in the other direction.

Each device generates a random number, designated in the specification as *ra* and *rb*. They each calculate a confirm value, *Ca* and *Cb* respectively. The calculation of *Ca* involves device A's public key *PKa* and its random number, *ra*. The calculation of *Cb* involves device B's public key *PKb* and its random number, *rb*. Note, there is no requirement for device A and device B to perform these calculations at the same time.

Device A now sends the device address that will be used when in-band pairing proceeds together with *ra* and *Ca* to device B. Device B sends its corresponding data values back to Device A.

A device address, random number, and confirm value constitute the OOB authentication data. As stated, authentication data may be provided by both devices or only one of the two devices.

Remember that this OOB exchange does not (by definition) involve Bluetooth. It might, for example, be accomplished with NFC, QR codes, or some other approach.

2.3.3.2 The In-Band Part

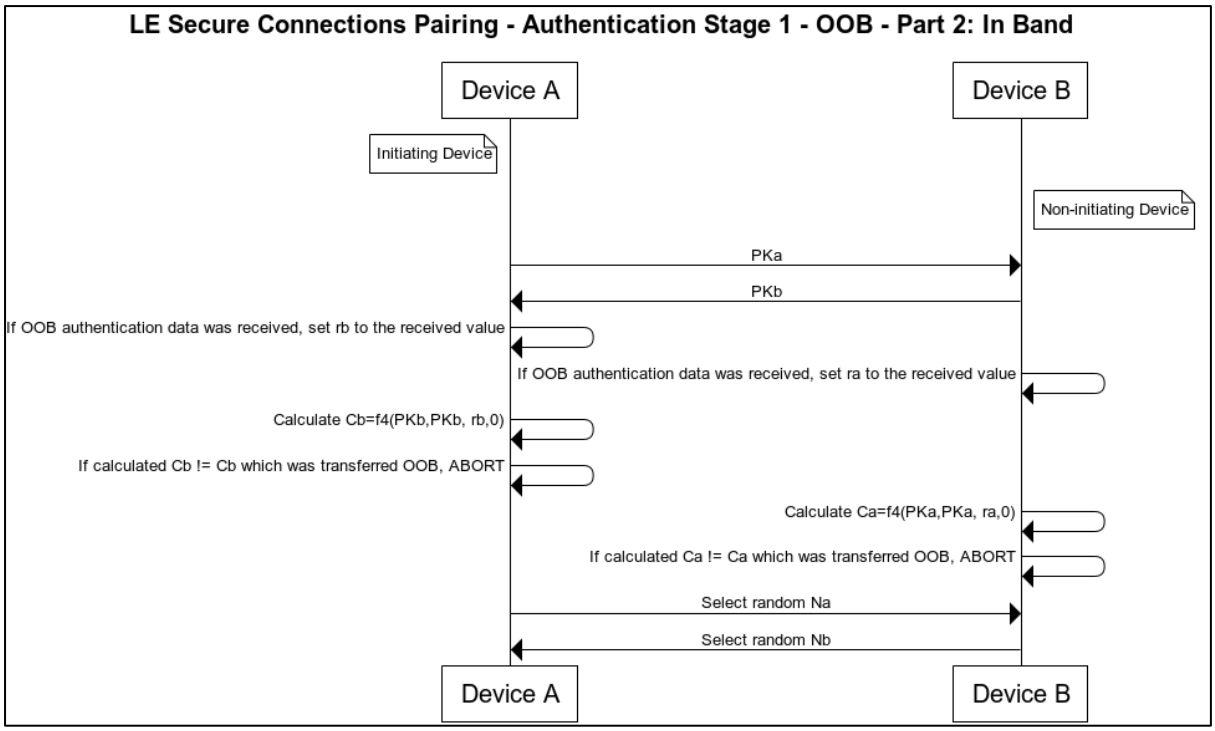


Figure 14 - The In Band part of LE Secure Connections Pairing OOB

Pairing now proceeds with pairing feature exchange, public key exchange, and the calculation of DHKey.

Next, one or both of the received confirm values *Ca* and *Cb* are recalculated using the public keys received in-band and the random numbers *ra* and *rb* received out of band. The calculated confirm values must match those received OOB. If either do not, pairing is aborted.

Authentication Stage 1 ends with each device calculating a nonce, *Na* and *Nb,* and these values being exchanged.

2.4 Authentication Stage 2

In Authentication Stage 2, further checks are made to ensure that the exchange of public keys, random numbers, and device addresses was completed correctly in the earlier steps. Note, this stage proceeds in the same way regardless of the association model that was used in Authentication Stage 1.

There are four steps in Authentication Stage 2.

1. The Long Term Key (LTK) and a key called the MacKey are calculated using function f5
2. A check value called *Ea* is calculated by device A and sent to device B where it is recalculated and compared with the received value
3. A check value called *Eb* is calculated by device B and sent to device A where it is recalculated and compared with the received value
4. Encryption is started on the link using the calculated LTK in readiness for phase 3

If either of the checks in steps 2 or 3 fail, pairing is aborted.

2.4.1 Calculation of MacKey and LTK

Each device calculates two keys, the LTK and a key called the MacKey using a function, *f5*.

```
MacKey || LTK = f5(DHKey, Na Nb, BD_ADDR_C, BD_ADDR_P)
```

Note the arguments used: the DHKey shared secret, the two random numbers *Na* and *Nb* that were exchanged during Authentication Stage 1, and the addresses of devcies A and B, named in the formula as BD_ADDR_C and BD_ADDR_P. Remember that the derivation of DHKey involved each device's private key early on in the pairing process and that private keys are not disclosed to the other device.

About function f5:

f5 is the *LE Secure Connections key-generation function*. It uses the AES-CMAC message authentication code algorithm with DHKey as its 128-bit key.

2.4.2 Check #1

Device A now calculates a check value called *Ea* and sends it to device B. Device B *should* have the same values available to it as a consequence of earlier stages of the pairing process. It recalculates *Ea* using those argument values and compares the result with the *Ea* value received from device B. They should be the same. If they are not, pairing is aborted.

The calculation of *Ea* is accomplished using a function called *f6*. This function uses AES-CMAC with the MacKey as the key. Remember that the derivation of MacKey involved DHKey, which itself involved the devices' private keys. Private keys are never disclosed, so this check will fail if an attacker device is attempting to trick another device into pairing with it.

About function f6:

F6 is the LE Secure Connections check value generation function. It uses the AES-CMAC message authentication code algorithm with MacKey as its 128-bit key.

2.4.3 Check #2

If check #1 was successful, the process is repeated but this time with device B calculating a check value, *Eb*. *Eb* is sent to device A where it is recalculated and compared with the received value. If they are not the same, pairing is aborted.

2.4.4 Start Link Encryption

The final step in Authentication Stage 2 is to start encryption on the link using the LTK to create a session key.

Phase 3 – Key Distribution

This phase is common to both LE Legacy Pairing and LE Secure Connections pairing, although the types of key that may be distributed are not the same for the two methods.

When LE Legacy Pairing is in use, LTK, IRK, and CSRK may be distributed (depending on the key requirements determined in phase 1). In addition, values called EDIV and RAND, which will act as identifiers for the LTK in the security database, are distributed.

When LE Secure Connections is in use, IRK and CSRK may be distributed during this phase. LTK has already been exchanged during phase 2. EDIV and RAND are not used when LE Secure Connections was used for pairing and, instead, the Bluetooth device address is used as the LTK identifier in the security database.

In the case of LE Secure Connections, for phase 3, the link has been encrypted using a session key derived from the LTK derived in phase 2. In the case of LE Legacy Pairing, STK is used to create the session key with which to encrypt the link at this point.

Recap – A Summary of LE Secure Connections Pairing

Phase 1 – Decides that LE Secure Connections pairing should be used by looking at the IO capabilities of the two devices and other pairing requirements that the devices express in the pairing feature exchange procedure.

Phase 2 – Authentication stage 1.

> **Step 1**: The two devices possess Elliptic Curve public-private key pairs. They exchange their public keys during this phase. Their private keys are never disclosed.
>
> **Step 2**: A shared secret called the DHKey is calculated by both devices using their own private keys and the public key of the other device, which is received in step 1.
>
> **Step 3**: A confirmation procedure takes place using public keys and nonce values, exchanged in a strict sequence. This involves the Responder calculating and disclosing to the Initiator, a confirmation value that cannot be later changed (for example, by an attacker).
>
> **Step 4**: If MITM protection has been requested, the user is also involved in an interactive procedure by which the user confirms that the device they are pairing with is the device they think they are pairing with (i.e. devices are authenticated by the user).
>
> **Step 5:** Authentication Stage 2 – The devices authenticate each other using a procedure that involves the private keys of the two devices. A key called the Long Term Key (LTK) is calculated.

Phase 3 – The link is encrypted with a session key derived from LTK, and keys of types established in phase 1 are distributed. Sometimes this is a unidirectional process from Peripheral to Central and sometimes it is bi-directional.

Devices are now paired. If the LTK/CSRK/IRK keys are stored for later reuse, they are said to be bonded.

An Appraisal of Bluetooth Pairing and Security

As stated in the introduction, device manufacturers need to assess their security requirements and this, coupled with device capabilities and constraints, should lead to the selection of appropriate Bluetooth security features, including the approach to be taken when pairing. But what recommendations can be made, and what would be regarded as best practice?

LE Secure Connections is Best

Unless there is an issue preventing its use, LE Secure Connections should always be used for pairing, coupled with an appropriate association model.

LE Secure Connections is considerably more secure than the old LE Legacy Pairing method. It is secure against passive attacks, whereas LE Legacy pairing is not (in all cases other than when using the OOB association model). This is due to the low entropy in TK and the role TK plays in key distribution. By observing the exchange made during pairing using a sniffer and then launching a brute force attack on TK, you could obtain all of the exchanged keys.

Provide MITM Protection

Unless there is an issue preventing it, MITM protection should always be used, and user authentication of devices should be a part of the process. If MITM protection is not used, the connection is susceptible to being intercepted and modified by an attacker during pairing.

OOB Authentication May Be the Most Secure Way to Use LE Legacy Pairing

If you must use LE Legacy Pairing, perhaps because old devices must be supported, then OOB authentication can provide the highest entropy (a full 128 bits) in the establishment of TK and, therefore, offer the greatest protection. But do not forget, the OOB channel used for sharing the OOB authentication data must also be secure, so this must also be assessed.

Use the Largest Encryption Key Sizes Allowed

Encryption key size is negotiated during pairing and will constrain the size of the LTK created during LE Legacy Pairing. If you are using LE Legacy Pairing and have control over the design of both devices to be paired, ensure that the maximum key size of 16 octets is used where possible. Furthermore, if your application requires high security (e.g., medical devices where tampering is a potential life-and-death problem), then configure your device to reject pairing and close the connection with any peer device that uses a key size that is less than 16 octets.

Link Encryption

Devices that have paired and exchanged an LTK may use encryption at the link layer to encrypt packets exchanged when in a connection.

Starting Link Encryption

The host on the Central device will instruct its controller to start link encryption when a connection is initially established if the security mode and level indicate that this is required. It will also start link encryption if access to a GATT attribute that has the encryption permission assigned to it is attempted if link encryption has not yet been started. A later section looks at attribute permissions.

Note that link layer encryption can only be applied to data when in a connection. Therefore, it is not possible to use this security feature with connectionless communication.

Starting encryption may be accomplished by the host sending the controller an HCI_LE_Start_Encryption command (on devices where the optional HCI is implemented).

The Peripheral may request security by sending the Central an SMP Peripheral Security Request PDU. The Central may respond to the request by pairing, if this has not already been done, or simply by starting encryption on the link.

Figure 15 shows a trace of the PDUs exchanged when a Central device, not yet paired with the connected Peripheral, attempts to read an attribute that has set the encryption permission. This permission indicates that the attribute may only be read if an encrypted link has been established.

Figure 15 - Starting encryption during pairing after an Insufficient Encryption error

Since this has not yet happened, and since it can only happen if the two devices have paired, pairing is initiated. In this case, since LE Legacy Pairing is being used, the link becomes encrypted with a session key derived from the STK.

Once established, the encrypted link continues to be used until it is terminated.

The e*ncryption start procedure* involves a number of steps. PDUs exchanged by Central and Peripheral are shown in Figure 15. Central and Peripheral each generate part of an initialisation vector value (*IV_C* from the Central, *IV_P* from the Peripheral) and part of a session key diversifier (*SKD_C* from the Central device and *SKD_P* from the Peripheral). These are exchanged in an

LL_ENC_REQ PDU sent from the Central to the Peripheral and a LL_ENC_RSP PDU that the Peripheral replies with. Each device then concatenates *IV_C* and *IV_P* to create the full IV value and, similarly *SKD_C* and *SKD_P,* to create the full session key diversifier. Each device is now in possession of the same, complete IV and SDK values.

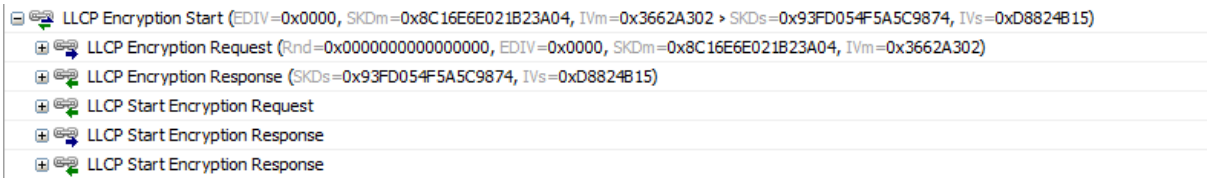


Figure 16 - Encryption Start

The hosts in both Central and Peripheral provide their link layers with the LTK to be used by sending the HCI_LE_Start_Encryption command. *Responding* (as opposed to *initiating*) devices determine the LTK to select from their security database using EDIV and RAND values as an identifier if LE Legacy Pairing was used or, the initiating device's Bluetooth address, if LE Secure Connections pairing was used.

A session key is now calculated by encrypting the SKD with the LTK as the key, as shown in Figure 17.

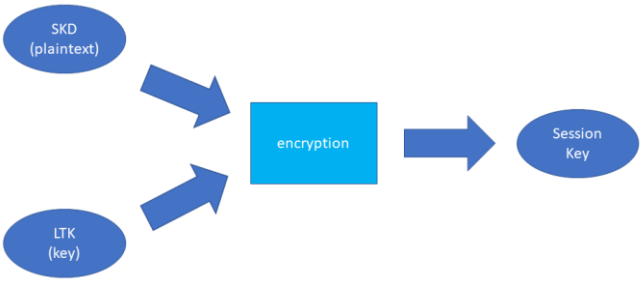


Figure 17 - Session Key creation

Next, the Peripheral sends an *unencrypted* LL_START_ENC_REQ PDU to the Central. The Central responds with an *encrypted* LL_START_ENC_RSP PDU. The Peripheral responds with its own, encrypted LL_START_ENC_RSP PDU. You can see the two response PDUs in Figure 16, completing the encryption start procedure.

The link is now encrypted.

Encryption and Authentication of Link Data

Bluetooth LE encryption uses AES-CCM.

AES is the Advanced Encryption Standard and a widely used block cipher for encryption. See https://csrc.nist.gov/glossary/term/AES

CCM provides authentication by calculating a Message Authenticating Code (MAC). See https://csrc.nist.gov/glossary/term/AES_CCM

So, AES-CCM is a block cipher mode that both encrypts and authenticates data.

Note that in communications systems, the acronym MAC usually means Media Access Control. For this reason, the Bluetooth Core Specification instead uses the term Message Integrity Code or MIC for the CCM-calculated authentication code.

When encryption is enabled, all PDUs have a MIC appended to them before the Cyclic Redundancy Check (CRC) field. The data PDU's payload and MIC are then encrypted.

The calculation of the MIC value involves a number of inputs, including a nonce whose value includes a packet counter. This packet counter is 39 bits in length, and it is incremented for every non-empty packet transmitted.

The receiving device decrypts and checks the MIC for evidence of corruption or tampering. If the MIC check fails, the connection is terminated.

Figure 18 - Encrypted PDU layout

Verifying the On-Going Integrity of the Encrypted Link

LE Ping is a link layer control procedure for which support is optional. See Bluetooth Core Specification 5.2 Vol 6 Part B section 5.1.8.

Where supported, LE Ping allows the link layer in a local device to check that a remote device, with which there has not been any communication for some time, is still present, still able to generate valid encrypted packets, and that no packets have been lost.

The purpose of the LE Ping procedure is to provide a mechanism that can be used to detect attacks that work by deliberately suppressing legitimate Bluetooth messages.

To use *LE* Ping, the host sets a link layer parameter called Authenticated_Payload_Timeout. This parameter indicates the maximum amount of time that may pass in between the receipt of packets that are authenticated by a valid MIC. The link layer sets a timer to monitor this condition and, when the specified Authenticated_Payload_Timeout time frame is close to being exceeded without having received an MIC-authenticated packet from the remote device, the link layer invokes *LE Ping*.

LE Ping involves notifying the local host that an authenticated payload timeout event has occurred and the local link layer sending a LL_PING_REQ PDU to the remote link layer. If the remote link layer supports the *LE Ping* procedure, it shall reply with an LL_PING_RSP PDU with a valid MIC.

If one or more messages have been suppressed, the MIC in the LL_PING_TSP_PDU will fail verification because the packet counter used in calculating the MIC shall not be contiguous with the packet counter related to the MIC from the last packet received.

If for any reason, the remote device cannot provide the required verification, the local device shall drop the link.

Pausing and Resuming Encryption

There may be occasions where the encryption key used to secure the link needs to be changed. For example, when pairing and preparing to encrypt the link using the distributed LTK it is found that the link in use has already been encrypted, then the link layer needs to switch from using the current key to the LTK.

To allow the changeover to take place without disconnecting the current link, a link layer procedure called the Encryption Pause procedure is defined. See Vol 6 Part B section 5.1.3.2 of the Bluetooth Core Specification version 5.2. While encryption is paused, unencrypted data PDUs shall not be sent.

At the end of the pause procedure, encryption of the link is resumed using the new key.

Privacy

Private Addresses

Bluetooth devices use an address called the Bluetooth device address (BD_ADDR) as an identifier. This address appears over the air in various PDUs, including advertising PDUs, in a field called AdvA or, in the case of directed advertising, TargetA. Addresses can also appear over the air when performing active scanning in a field called ScanA.

There is a risk that through scanning for and collecting such addresses over time, a malicious third-party could build a picture of the movements of devices and, by inference, the human users of those devices. The Bluetooth privacy feature is designed to mitigate this risk.

When using the privacy feature, devices have two addresses. The first is the *identity address,* which acts as an unchanging identifier of the Bluetooth device. The second is the *private device address*, which changes periodically.

Private addresses disguise the identity of a device and when in use, it is the private address which is disclosed in over the air packets, not the identity address.

The host determines whether to use privacy and when. When actively using privacy, the device is said to be in *privacy mode*. Privacy mode may be enabled by the host sending an HCI_LE_Set_Privacy_Mode command to the controller.

The host is responsible for informing the controller how often it should generate a new private address. It does this using the HCI_LE_Set_Resolvable_Private_Address_Timeout command that allows a timeout value of between 1 second and 11.5 hours to be specified. The link layer in the controller uses this parameter to establish a timer. When the timer expires, a new private address is generated. The Bluetooth Core Specification recommends a value of 15 minutes for private address regeneration (ref: Bluetooth Core Specification 5.2 Vol 6, Part B section 6.1).

There are two types of private device address: *resolvable* or *non-resolvable.*

If device A is using *resolvable private addresses*, was paired with device B, and the IRK (Identity Resolution Key) was distributed during pairing, IRK can be used by device B to *resolve* the private address of device A to its corresponding identity address. In other words, when resolvable private addresses are in use, devices that have paired and exchanged IRKs can determine the true identity address of a device from its private address. Other devices that have not been paired, cannot do this and simply see a series of different BD_ADDR values over the air over time, and have no way of knowing that these address values are being transmitted by the same physical device.

When *non-resolvable private addresses* are in use, not even paired devices can determine the corresponding identity address, so privacy is protected even from trusted devices that they have been paired with.

Privacy Modes

Two privacy modes are defined. A privacy mode applies to a specific peer device with which the local device has previously paired and exchanged IRKs with.

The default privacy mode is called *network privacy mode*. When this mode has been selected for a given peer device, any advertising packets from that device which contain the public identity address shall be ignored by the link layer and not processed further up the stack.

The other mode is called *device privacy mode*. In this mode, the local device will accept both private addresses and public identity addresses in packets from the applicable peer device (the one for which *device privacy mode* has been set).

The Resolving List

Information concerning peer devices with which IRK has been exchanged, including their IRK, identity address and privacy mode are maintained by the host in a list of records called the *Resolving List*. The list is passed, in part or in full, from the host to the controller so that the link layer can consult it.

Private Address Generation

A *non-resolvable* private address is simply a 48-bit address with the two most-significant bits set to zero and the value of the remaining 46 bits set to a random number. At least one bit of the random number must be set to 0 and at least one must be set to 1. The private address may not be the same as the public (identity) address, for obvious reasons.

Figure 19 - Format of a non-resolvable private address

To generate a *resolvable* private address, the generating device must possess either its own IRK (the *Local Identity Resolving Key*) or the IRK of the peer device (the *Peer Identity Resolving Key*). Devices which have paired and exchanged IRKs will by definition possess each other's IRKs.

There are two parts to a resolvable private address. The 24 most significant bits represent a 24-bit random number known as *prand*, which has had its two most significant bits set to 0 and 1, respectively. The 24 least significant bits of the address are set to a hash value which is calculated using a function named *ah*, which takes an IRK and *prand* as its input parameters. The function *ah* is defined in the Bluetooth Core Specification 5.2 in Volume 3, Part H, Section 2.2.2.

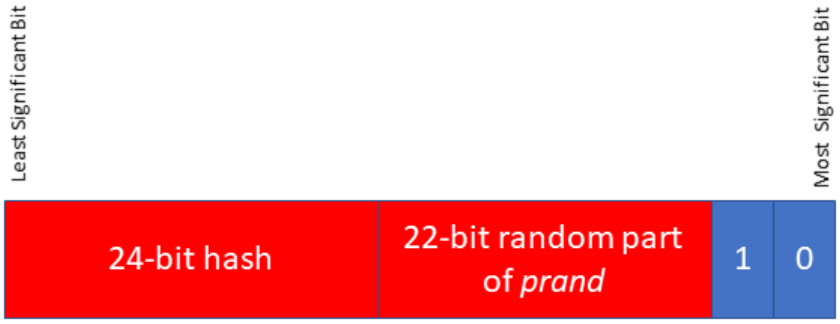


Figure 20 - Format of a resolvable private address

Private Address Resolution

An address will be recognized as being a resolvable private address if the two most significant bits of the address are set to 0 and 1 respectively. The receiving, local device may proceed to attempt to resolve the private address back to the corresponding identity address of the sending, peer device as follows:

For each IRK known to the local device;

Calculate a hash of address' *prand* value using the *ah* function;

```
localHash = ah(IRK, prand)
```

Compare the calculated local hash with the hash value contained within the received private address. If they match, then the IRK is the key which was used to create the private address, and the *resolving list* will contain the associated identity address of the peer device. The address has been resolved, so the procedure may now terminate.

Note that private address resolution may be performed in either the host or the controller. Resolving private addresses in the controller has the advantage of being more power efficient and allowing link layer device filtering via the *accept list* to be utilized.

Attribute Permissions

Overview

The Generic Attribute Profile (GATT) provides a means by which device data and capabilities may be represented in a hierarchical structure consisting of GATT services, characteristics, and descriptors. Each of these three types of GATT construct is an instance of something more general, called an *attribute*. Attributes are declared and defined in an *attribute table,* which flattens the hierarchical structure and assigns each entry in the table a unique identifier known as a *handle*.

Connected clients access the attributes in the attribute table using a protocol called the Attribute Protocol (ATT) using a variety of operations, such as characteristic value reads and writes. Servers may send data to connected clients using GATT procedures, such as *characteristic value notifications* and *characteristic value indications*.

Each attribute in the *attribute table* includes a set of *attribute permissions*. Attribute permissions define various rules regarding what kind of access a remote client may or may not have to that

attribute (e.g. the ability to read its value) and any conditions that might apply before that access is granted. For example, an attribute's permissions might stipulate that clients can read its value but only over an authenticated and encrypted link. Attribute permissions also apply to ATT servers and their communication with clients using notifications and indications.

The attribute permissions system gives Bluetooth profile designers fine control over the security rules that apply to the data and device capabilities exposed by GATT to connected clients. A specification called a *profile* usually defines the GATT services, characteristics, and descriptors for a given use case, including their respective attribute permissions.

Designers may design their own custom services, characteristics, and descriptors and should ensure that the security requirements of each is carefully considered, specified, and implemented.

Securing access to data items is a common requirement in many fields of software development. Assessing the value of data and the possibilities for its misuse should be standard practice in any field. But with a Bluetooth connectable device, quite often attributes correspond to the physical capabilities of the device and insufficient or inappropriate security can have significant consequences. For example, writing a value of 1 to a particular characteristic might have the intended consequence that a machine switches on and starts to operate. It does not take a lot of imagination to consider cases where the most-secure form of pairing and the most-strongly authenticated and encrypted connection must be in place before a client device should be allowed to do this.

The Core Specification defines four distinct types of attribute permission.

Access Permissions

This permission type defines whether or not an attribute may be read, written to, or both read and written to by a connected client.

Encryption Permissions

This permission type indicates whether or not access to the associated application must only be granted when an encrypted link is in use or, alternatively, that no encryption is required.

Authentication Permissions

This permission type indicates whether or not access to the associated attribute must only be granted when the client device was authenticated when it was paired, using an appropriate pairing association model.

Authorization Permissions

Authorization permissions indicate whether or not a client must obtain authorization before being allowed to access the attribute. But what does this mean?

This permission type allows the implementation of all manner of miscellaneous rules, including ones that augment or refine those expressed by other permission types, such as the authentication or encryption permissions. For example, the encryption permission may indicate that access is only allowed over an encrypted link. This could be refined to state that a minimum negotiated key length is also required. Or, if authentication is required, we could go further and indicate the type of

authentication that must have been performed when pairing, before access to this attribute is allowed.

Authorization permissions can be quite independent of the other permission types, though. We could stipulate a rule that during the lifespan of a connection, a given characteristic may only be read after some other characteristic has first been written to. Authorization permissions are very flexible and offer designers of profiles the ability to specify very precise security rules for each of the attributes in the profile.

Permission Representation and Implementation

How attribute permissions are represented in the attribute table, and how authorization rules are evaluated, is an implementation-specific issue. The Bluetooth Core Specification does not mandate a particular approach.

Permission Failures

If an attempt to access an attribute is made, and the conditions of the associated attribute permissions are not met, the attribute protocol defines a number of responses that may be returned to indicate to the client device that the access request was denied and for what reason. Responses of this sort include *insufficient encryption*, *insufficient authentication,* and *insufficient encryption key size*.

Data Signing

Devices that have paired and exchanged CSRKs may use *connection data signing*. This is a procedure that appends a digital signature to a particular ATT PDU, the *Signed Write*. This is a rarely used procedure because there is a better alternative but is included here for completeness.

A device, acting as an ATT server and receiving a Signed Write, recomputes and verifies the signature by comparing the computed value with the signature attached to the ATT PDU. If the two values match, the signature has been verified and integrity and authenticity of the ATT PDU checked. If they do not, then the authentication check has failed, and it is recommended that the ATT server closes the connection.

Data signing may only be used with an *unencrypted* link. Its use is referred to as security mode 2, level 2.

The Bluetooth Core Specification calls the data signing procedure the *Connection Data Signing Procedure* because it can only be used in connection-oriented scenarios as opposed to connectionless ones.

The data that is signed consists of the ATT opcode, handle, and value concatenated with the string representation of a counter called the SignCounter. A SignCounter is maintained in the security database for each paired device with which CSRK has been exchanged. It is set to zero whenever CSRK is generated and incremented for each ATT Signed Write sent. The purpose of SignCounter in data signing is to protect against replay attacks.

Note that there is only one user of the Connection Data Signing Procedure, namely the aforementioned Attribute Protocol Signed Write.

Comment: The original intention of security mode 2 was to support specialized use cases on devices with low computing power, where fast and authenticated connection setup using directed advertising was required. Authentication of the connecting device was to be achieved by sending a single Signed Write ATT PDU over the connection as soon as it had been established. In the use cases that were considered, there was no requirement for safeguarding the confidentiality of data over the connection afterwards.

Bluetooth LE link encryption adds a Message Integrity Check field to all PDUs and so provides for both the authentication and confidentiality of PDUs.Therefore, signatures on Signed Write commands are not checked when sent over an encrypted link because the MIC is deemed to have provided the required authentication.

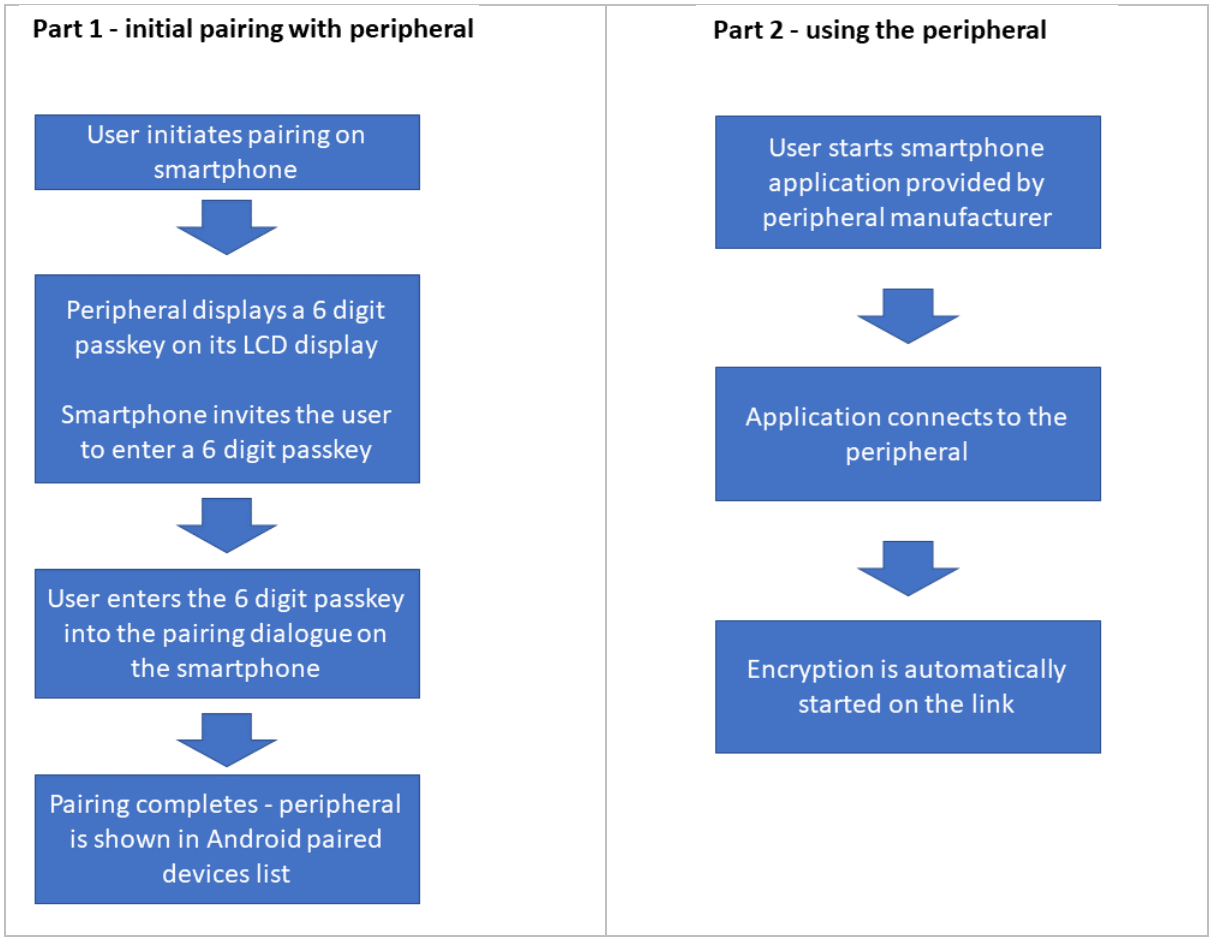
These days, there is no good reason to use Connection Data Signing or security mode 2, level 2. If you have a requirement to authenticate messages, you should use security mode 1 with level 2, 3, or 4 (i.e. link encryption) instead.

4.3 Orientation

In this section, a number of scenarios involving Bluetooth LE security features are presented. The intention at this stage is to put some of the features introduced in the Overview section, and reviewed in the Examination section, into context to help with orientation.

Orientation Scenario 1 - Pair Then Use

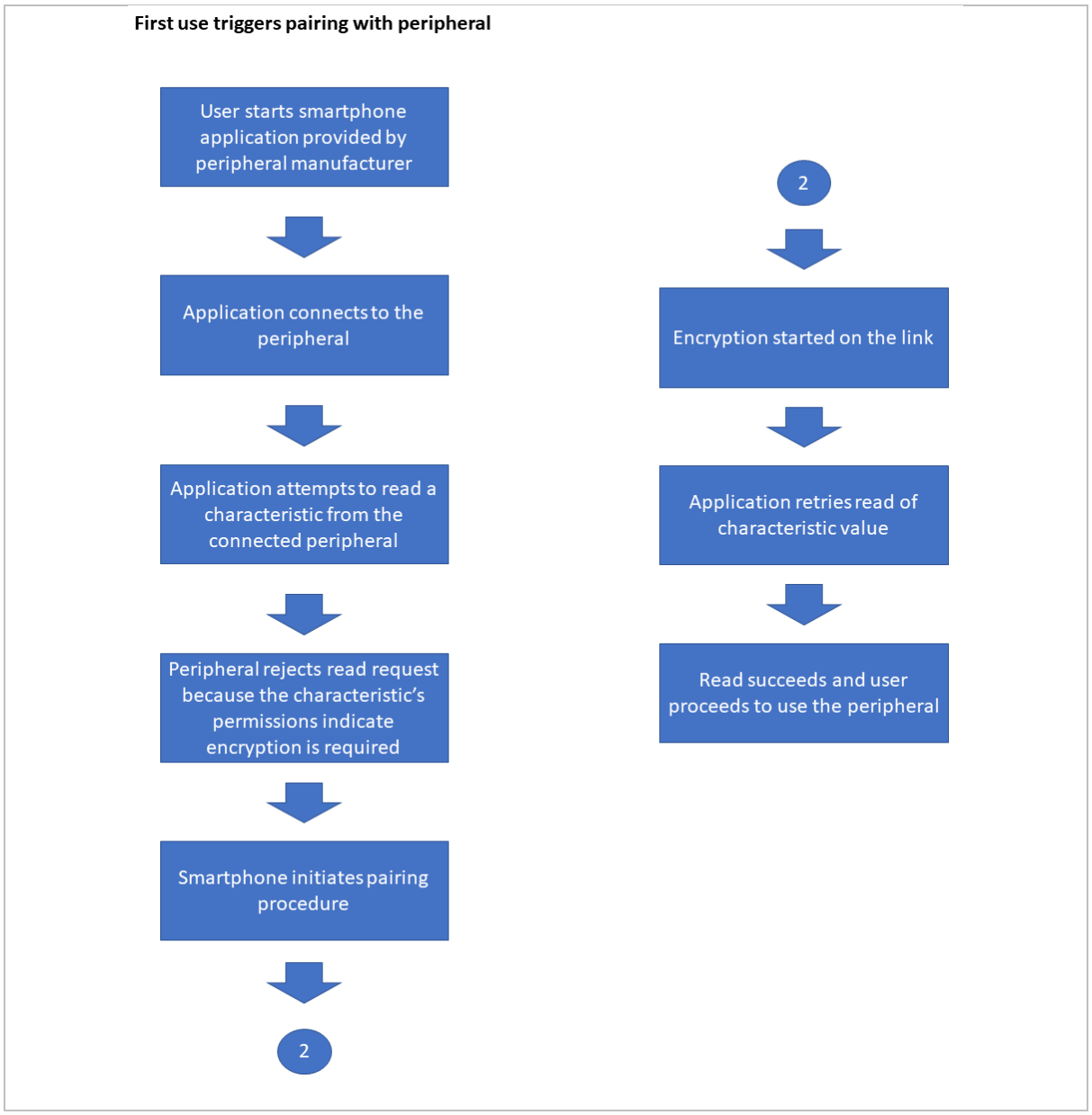
A user has an Android smartphone they wish to use with a Bluetooth peripheral device which requires security mode 1 with security level 3 (encryption with authentication). In the first step, users pair their phone with the peripheral device from the Android Settings/Bluetooth screen. Sometime later, they use a smartphone application to connect to the peripheral device and use it.



Pairing provided each device with an LTK key to use for encryption, and the key was stored by each device (i.e. they became *bonded)*. Later, when the smartphone connected to the peripheral, due to the security mode and level requirements, the connection was immediately encrypted using the LTK shared during pairing.

Orientation Scenario 2 - Use and Pair

Users have an iOS smartphone they wish to use with a peripheral device that requires security mode 1 with security level 2 (encryption required). They use a smartphone application to connect to the peripheral device in order to use it. The peripheral's attribute table contains a characteristic that specifies that encryption is required in its attribute permissions. When the smartphone application attempts to read this characteristic, the read fails due to the encryption permission not being satisfied because the link is not yet encrypted, and pairing is automatically triggered. After pairing, the read operation succeeds.

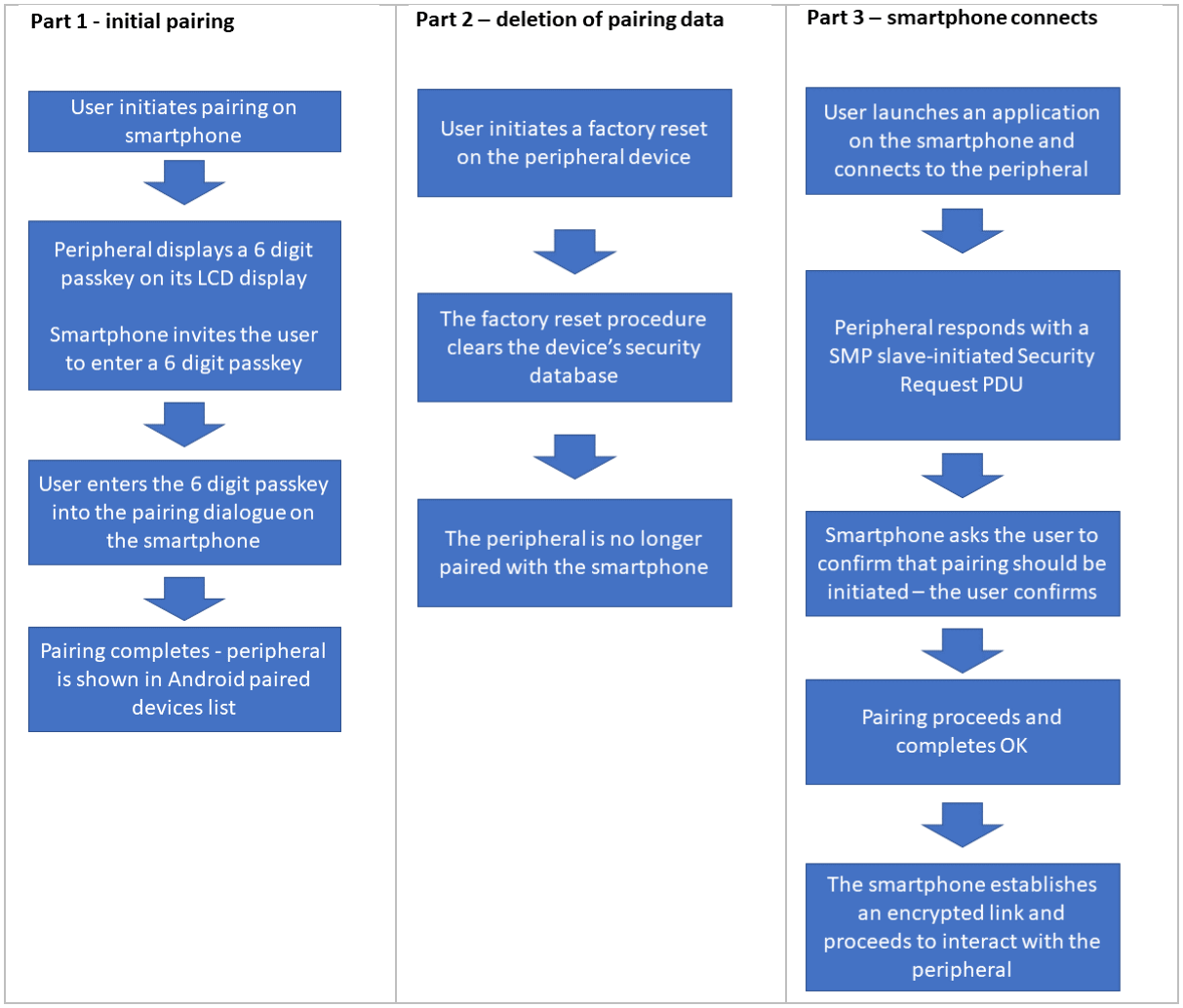


Orientation Scenario 3 - Pairing Relationship Deleted

Users have smartphones that they have previously paired with a peripheral device. To resolve an issue following the manufacturer's instructions, users perform a factory reset on the peripheral. Unbeknownst to the users, this action clears a peripheral's security database and thus, all pairing data. The users' smartphones are still paired with their peripherals, but the peripherals are no longer paired with any other device.

The peripheral device has been programmed to operate in LE security mode 1, level 4 (*Authenticated LE Secure Connections pairing with encryption using a 128-bit strength encryption key*).

When a user next connects their smartphone to the peripheral using an application, the peripheral responds by sending an SMP Security Request PDU. The smartphone responds by asking the user to confirm that pairing should be performed. The user confirms the action, pairing proceeds, and the smartphone and peripheral are once again paired and able to set up the encrypted connection.



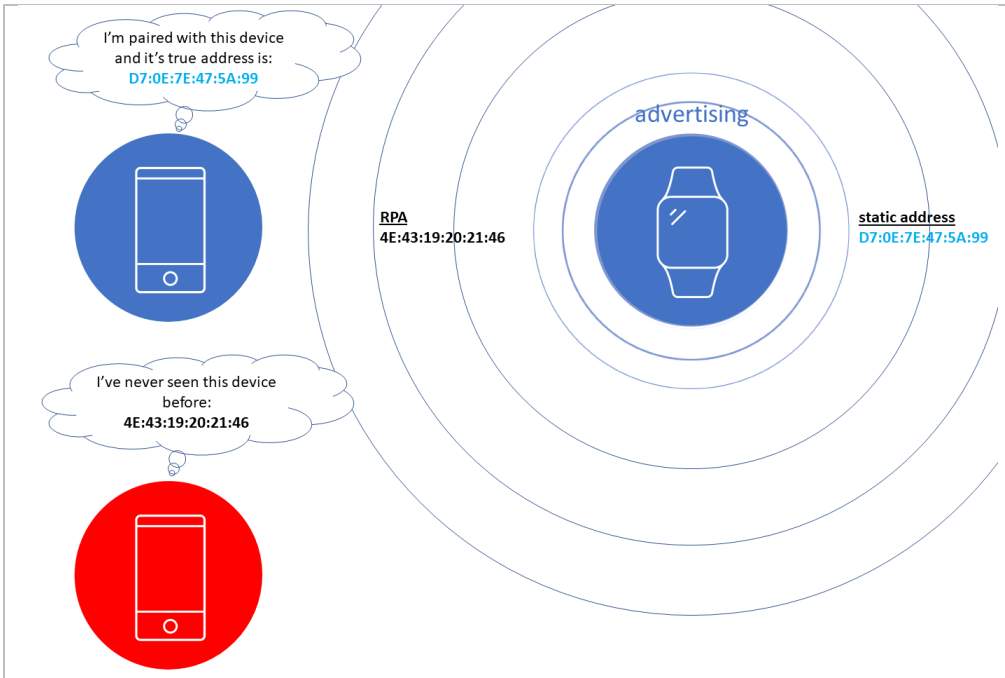
Orientation Scenario 4 - Privacy

Users have wearable activity trackers and smartphones that they use with the wearable devices. Most of the time, the activity tracker and smartphone are not connected. The activity tracker gathers data about a user's physical activities using its sensors and stores the data locally. It also transmits Bluetooth advertising packets so that a smartphone can discover it, connect, and collect the most recently accrued activity data whenever a user runs a particular application on the smartphone.

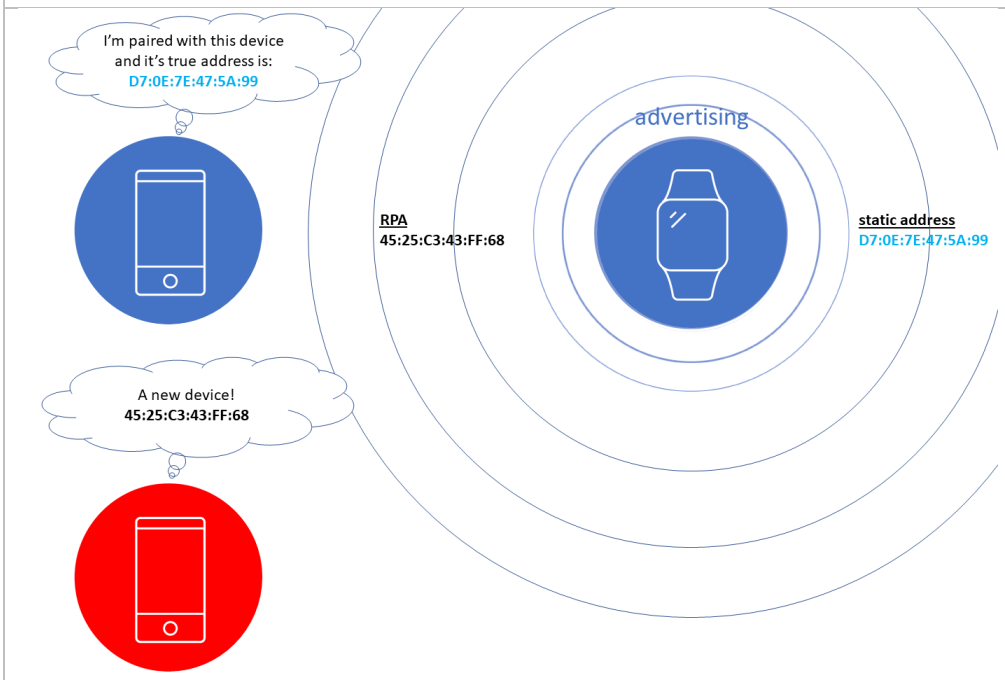
The activity tracker's product designer has recognized that Bluetooth advertising could represent a privacy issue and allow its user to be tracked. Consequently, the product designer wisely opted to use Bluetooth resolvable private addresses (RPAs) in advertising packets and, as such, the user has had to pair the activity tracker and smartphone so that address resolution can be performed by the trusted smartphone.

Other devices (malicious or otherwise) that scan for advertising packets have not been paired with the activity tracker and therefore do not possess the required IRK. As such, they cannot resolve the periodically changing private addresses in the tracker's advertising packets, and the user cannot be tracked using the advertised device address.

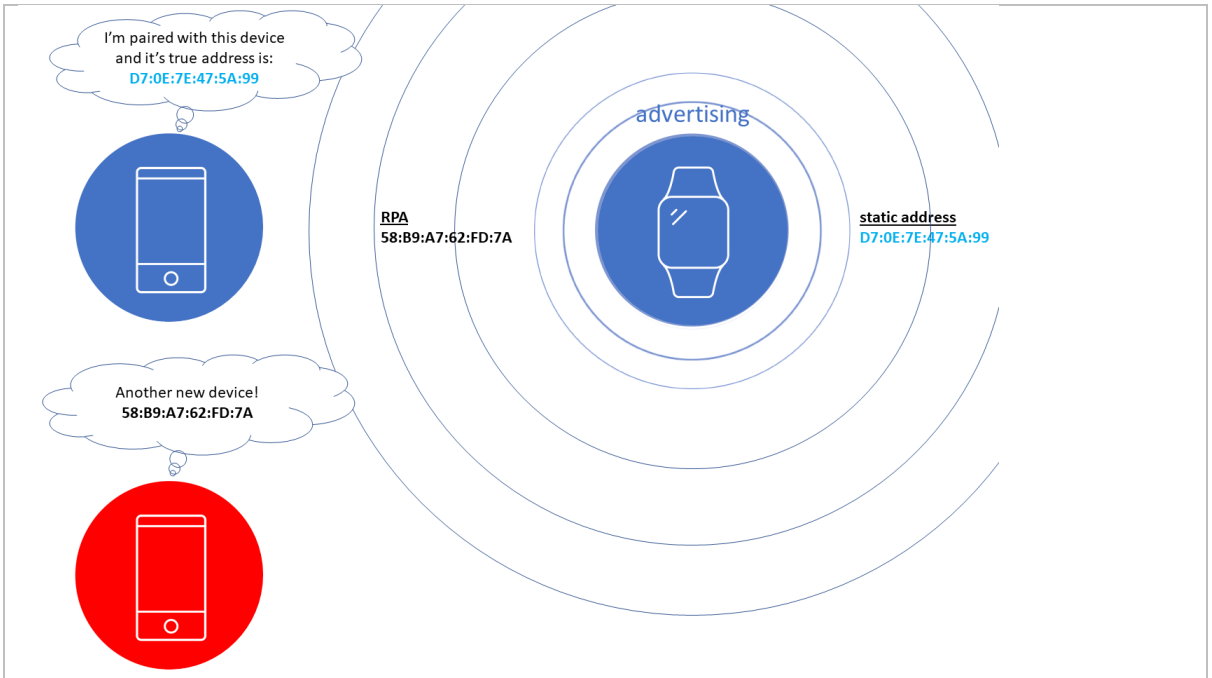
In Figure 21, the activity tracker has a static address of D7:0E:7E:47:5A:99 but is advertising resolvable private addresses that change every 10 minutes. The smartphone paired with the activity tracker is able to resolve the RPA. Another smartphone, which has not been paired, cannot resolve the RPA and so cannot uniquely identify the advertising activity tracker.



1. Initially, the activity tracker advertises with RPA 4E:43:19:20:21:46



2. After approximately 10 minutes, the advertised RPA changes to 45:25:C3:43:FF:68



3. Now, the advertised RPA has changed to 58:B9:A7:62:FD:7A

Figure 21 – Bluetooth LE Privacy in action

4.4 Exploration

It is now time to gain some hands-on experience and see how some of these security features can be made use of from code.

You will be using the Zephyr SDK, as mentioned in the *Software and Hardware Requirements* section. The sample code used in these exercises was developed against the *zephyr-v2.6.0* branch in GitHub. If you are new to Zephyr, make sure you following the Getting Started guide and set up your development environment properly.

In some of the following exercises, your job will be to modify source code so that it makes use of particular Bluetooth security features. In others, you will simply need to review the provided code to appreciate what was involved.

The code provided is intended to act solely as a set of examples that illustrate the types of step involved in making use of Bluetooth security with this particular platform and SDK. Other SDKs will differ from Zephyr in their approach, but perhaps not dramatically so. The Zephyr code provided in this resource should help illustrate not only the Zephyr approach, but the types of approach that are typical across SDKs for different platforms.

Note that if you decide to build and install the sample code, some of the projects require a board which has at least two buttons. In developing this study guide, a Nordic Semiconductor nRF52840-DK was used in testing and is an example of a suitable board. Build for this type of board with:

```
west build -b nrf52840dk_nrf52840
```

It is recommended that after each exercise you clear persisted data such as security keys from previous pairings from your developer board. The Nordic Semiconductor nRF52840-DK can be cleared using the Nordic Command Line Tools and the command *nrfjprog -e*.

4.4.1 No Security

Code

Create a Zephyr project called blessg_ex1. Copy the content of this resource's code\start_points\blessg_ex1 folder into your blessg_ex1 and create a sub-folder called build.

This code will turn your board into a GAP Peripheral and GATT/ATT server that includes a single, custom GATT service. The service contains two characteristics, one that may have an integer in the range 0 - 10 be written to it, which may be read, and which will return the value last written to the other characteristic. The characteristic that supports the Write Request will be referred to as the *input characteristic* since it allows us to input data to the peripheral device. The other characteristic will be referred to as the *output characteristic*.

Open the project folder in your favourite text editor and look at the services/custom_service.c source file. The array called *attrs[]* shown in Figure 22 defines the attribute table entries for the service followed by its two characteristics. The *attribute property* BT_GATT_CHRC_WRITE on the first indicates that it supports *attribute protocol write requests*. BT_GATT_CHRC_READ on the second means it supports *attribute protocol read requests*.

The next field in the BT_GATT_CHARACTERISTIC macro deals with *attribute permissions*. This is a bit mask field that, in this case, indicates an *access permission* of *writeable* for the first characteristic

and *read only* for the second. Importantly, note that the stated permissions **do not** stipulate that encryption, authentication, or authorization are required to read/write these characteristics. This means they may be accessed in the appropriate way without restriction.

```
static struct bt_gatt_attr attrs[] = {

 BT_GATT_PRIMARY_SERVICE(BT_UUID_CUSTOM_SERVICE),

 BT_GATT_CHARACTERISTIC(BT_UUID_INPUT, BT_GATT_CHRC_WRITE, BT_GATT_PERM_WRITE, NULL,
write_input, NULL),

 BT_GATT_CHARACTERISTIC(BT_UUID_OUTPUT, BT_GATT_CHRC_READ, BT_GATT_PERM_READ, read_output,
NULL, NULL),

};
```

Figure 22 - Attribute Table Definition

Test

Build this code and install it on your board. Make sure you specify the right board identifier to build for the board you are using. See https://docs.zephyrproject.org/latest/boards/index.html.

The output from these steps should look something like that which is shown in Figure 23.

```
C:\blessg_work\solutions\blessg_ex1>west build -b nrf52840dk_nrf52840
-- west build: generating a build system
-- Application: C:/blessg_work/solutions/blessg_ex1
-- Zephyr version: 2.6.0 (C:/workspaces/zephyr_source/zephyr), build: zephyr-v2.6.0
-- Found Python3: C:/python39/python.exe (found suitable exact version "3.9.2") found
components:  Interpreter
-- Found west (found suitable version "0.10.1", minimum required is "0.7.1")
-- Board: nrf52840dk_nrf52840
-- Cache files will be written to: C:\Users\mwoolley\AppData\Local/.cache/zephyr
-- Found dtc: C:/ProgramData/chocolatey/bin/dtc.exe (found suitable version "1.4.7",
minimum required is "1.4.6")
-- Found toolchain: gnuarmemb (C:/gnu_arm_embedded)
-- Found BOARD.dts:
C:/workspaces/zephyr_source/zephyr/boards/arm/nrf52840dk_nrf52840/nrf52840dk_nrf52840.dts
-- Generated zephyr.dts: C:/blessg_work/solutions/blessg_ex1/build/zephyr/zephyr.dts
-- Generated devicetree_unfixed.h:
C:/blessg_work/solutions/blessg_ex1/build/zephyr/include/generated/devicetree_unfixed.h
-- Generated device_extern.h:
C:/blessg_work/solutions/blessg_ex1/build/zephyr/include/generated/device_extern.h
Parsing C:/workspaces/zephyr_source/zephyr/Kconfig
Loaded configuration 'C:/blessg_work/solutions/blessg_ex1/build/zephyr/.config'
No change to configuration in 'C:/blessg_work/solutions/blessg_ex1/build/zephyr/.config'
No change to Kconfig header in
'C:/blessg_work/solutions/blessg_ex1/build/zephyr/include/generated/autoconf.h'
-- The C compiler identification is GNU 7.3.1
-- The CXX compiler identification is GNU 7.3.1
-- The ASM compiler identification is GNU
-- Found assembler: C:/gnu_arm_embedded/bin/arm-none-eabi-gcc.exe
-- Configuring done
-- Generating done
-- Build files have been written to: C:/blessg_work/solutions/blessg_ex1/build
-- west build: building application
[224/231] Linking C executable zephyr\zephyr_prebuilt.elf

[231/231] Linking C executable zephyr\zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      160740 B         1 MB     15.33%
            SRAM:       24294 B       256 KB      9.27%
        IDT_LIST:          0 GB         2 KB      0.00%

C:\blessg_work\solutions\blessg_ex1>west flash
-- west flash: rebuilding
ninja: no work to do.
-- west flash: using runner nrfjprog
Using board 683389225
-- runners.nrfjprog: Flashing file:
C:\blessg_work\solutions\blessg_ex1\build\zephyr\zephyr.hex
```

```
Parsing image file.
Erasing page at address 0x0.
Erasing page at address 0x1000.
Erasing page at address 0x2000.
Erasing page at address 0x3000.
Erasing page at address 0x4000.
Erasing page at address 0x5000.
Erasing page at address 0x6000.
Erasing page at address 0x7000.
Erasing page at address 0x8000.
Erasing page at address 0x9000.
Erasing page at address 0xA000.
Erasing page at address 0xB000.
Erasing page at address 0xC000.
Erasing page at address 0xD000.
Erasing page at address 0xE000.
Erasing page at address 0xF000.
Erasing page at address 0x10000.
Erasing page at address 0x11000.
Erasing page at address 0x12000.
Erasing page at address 0x13000.
Erasing page at address 0x14000.
Erasing page at address 0x15000.
Erasing page at address 0x16000.
Erasing page at address 0x17000.
Erasing page at address 0x18000.
Erasing page at address 0x19000.
Erasing page at address 0x1A000.
Erasing page at address 0x1B000.
Erasing page at address 0x1C000.
Erasing page at address 0x1D000.
Erasing page at address 0x1E000.
Erasing page at address 0x1F000.
Erasing page at address 0x20000.
Erasing page at address 0x21000.
Erasing page at address 0x22000.
Erasing page at address 0x23000.
Erasing page at address 0x24000.
Erasing page at address 0x25000.
Erasing page at address 0x26000.
Erasing page at address 0x27000.
Applying system reset.
Checking that the area to write is not protected.
Programming device.
Enabling pin reset.
Applying pin reset.
-- runners.nrfjprog: Board with serial number 683389225 flashed successfully.
```

Figure 23 - Building and Installing blessg_ex1

At this stage, your blessg_ex1 project is not using any Bluetooth LE security features. No security level is specified, and it will default to mode 1, level 1 (no security).

Launch the nRF Connect application on your smartphone and connect to your board. It will be advertising with a device name of “LVL1NONE” (defined in prj.conf). Write a value of 07 to the writable characteristic followed by reading the other characteristic. It should return 07 because this characteristic will always return the value most recently written to the other characteristic. You should not be required to pair.

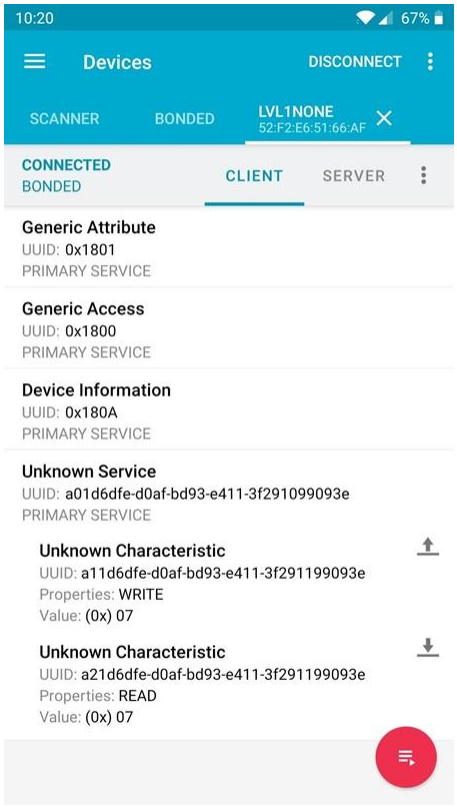


Figure 24 – Reading and Writing characteristics without constraint

If you have a sniffer, capture a trace. It should contain elements such as those depicted in Figure 25.

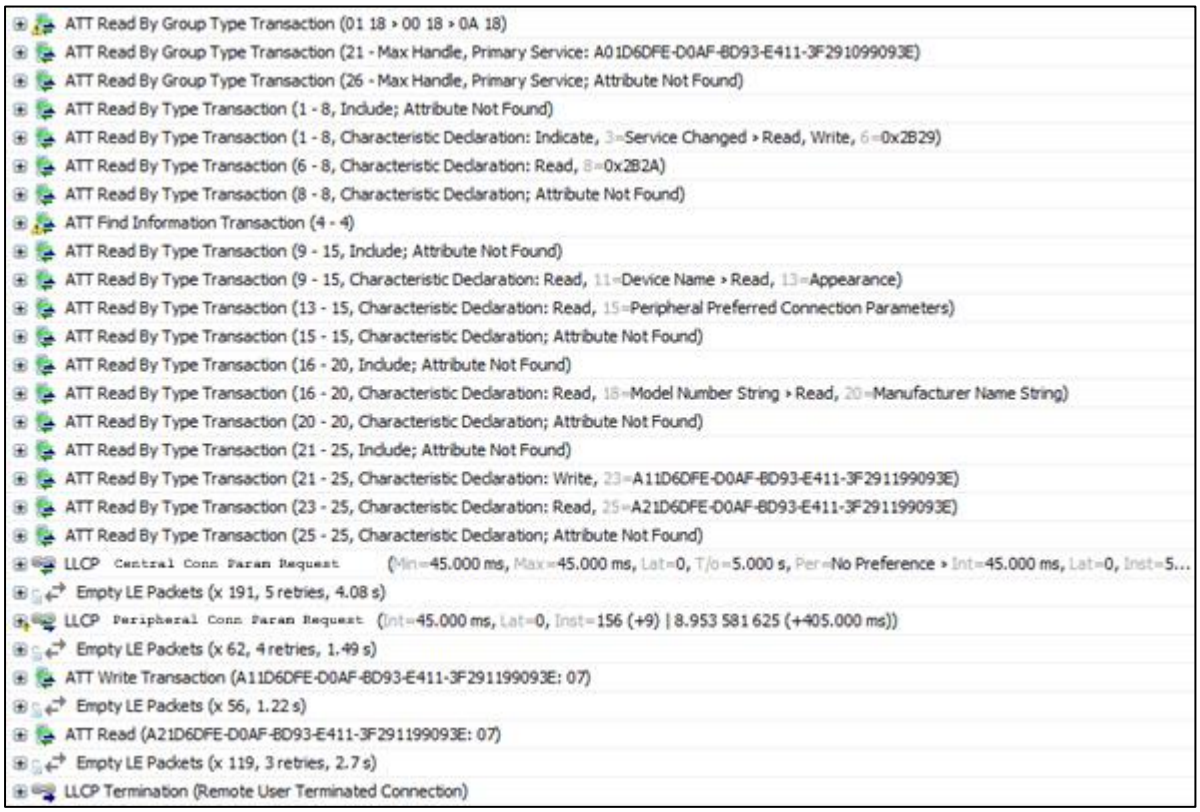


Figure 25 – Trace: Interactions with no security

The first set of ATT PDUs are the service discovery procedures being executed. Next, you will see an ATT Write followed by an ATT Read. Both are carried out successfully.

At no time do you see encryption being started on the link. This is as expected for this code.

4.4.2 LE Secure Connections Pairing

Your next task is to reprogram your board so that security mode 1, level 4 is selected. This will also be the only security level allowed, a mode that the Bluetooth Core Specification calls *Secure Connections Only Mode* (ref Vol 3 Part C section 10.2.4).

As such, LE Secure Connections pairing will need to be performed before any access to the characteristics in the custom service is possible.

Code

Create a Zephyr project called blessg_ex2. Copy the content of this resource's code\start_points\blessg_ex2 folder into your blessg_ex2 and create a sub-folder called build. All of the required code, except for that which is related to security, is already included in this source.

Your task now is to complete those sections marked with TODO comments so that your board implements security mode 1, level 4 and indicates IO capabilities and other pairing preferences that should result in LE Secure Connections pairing with the Numeric Comparison association model being used when you pair with your smartphone.

Open the file prj.conf and note the CONFIG_BT_SMP_SC_ONLY property. It is this property that selects *Secure Connections Only Mode*.

Also note the CONFIG_BT_TINYCRYPT_ECC property. It is this Zephyr build configuration property that enables support for Elliptic Curve Cryptography and, this in turn, causes the stack to indicate support for LE Secure Connections pairing.

Changes to the source code that are now required, fall into three areas:

1. Request security mode 1, level 4
2. Register callback functions relating to pairing and security level changes
3. Implement the callback functions

Proceed by adding the code fragments below in the appropriate places within the main.c source file, pausing to review the code and appreciate what it achieves.

Step 1 - Request Security Mode 1, Level 4

Update the *connected* function so it includes the call to *bt_conn_security* shown here:

```
static void connected(struct bt_conn *conn, u8_t err)
{
        if (err)
        {
        }
        else
        {
                printk("Connected\n");
                default_conn = bt_conn_ref(conn);

                // TO DO request mode 1 level 4 security
                int rc = bt_conn_security(default_conn, BT_SECURITY_L4);
```

```
        }
}
```

Step 2 - Register Callback Functions Relating to Pairing and Security-Level Changes

Declare a struct of type *bt_conn_auth_cb*. This struct is used to register three callback functions that will be involved in implementing pairing.

```
// TODO specify callbacks for pairing - IO Capabilities are implied from this struct
static struct bt_conn_auth_cb pairing_cb_display = {
                .passkey_display = pairing_passkey_display,
                .passkey_confirm = auth_confirm,
                .cancel = auth_cancel,
};
```

Now, modify the main function to register the three callback functions.

```
void main(void)
{

        printk("starting NCLEC\n");
        int err;
        configureButtons();
        err = bt_enable(bt_ready);
        if (err)
        {
                return;
        }

        clear_all_bonds();

        bt_conn_cb_register(&conn_callbacks);

    // TODO register callback for authentication step
        bt_conn_auth_cb_register(&pairing_cb_display);
}
```

Note the call to the clear_all_bonds() function. This has been included so that each time you reset your board, all bonds will be deleted so you can proceed to pair again. You would not usually include such a step but, since we're experimenting with pairing and you are likely to run this code a few times, it will make life easier.

Without it, you may see the following error message in the console when you try to pair the same smartphone more than once with your board without having cleared bonds from the previous pairing:

[00:00:22.104,522] <err> bt_hci_core: Failed to add IRK to controller

Update the conn_callbacks struct so that it includes a security_changed callback function. This function will be called when the security level has been changed to level 4 after pairing.

```
// TODO specify callback for security level changing
static struct bt_conn_cb conn_callbacks = {
                .connected = connected,
                .disconnected = disconnected,
                .security_changed = security_level_changed,
};
```

Step 3 - Implement the Callback Functions

Add the following callback function. It will simply write a message to the console to indicate when the security level has been changed and to what level. This will confirm you have switched to level 4.

```
// TODO implement security_level_changed
static void security_level_changed(struct bt_conn *conn, bt_security_t level, enum
bt_security_err err)
{
	printk("security_level_changed to %d\n", level);
}
```

Finally, add the following functions to implement the various pairing-related callback functions.

```
// TODO implement pairing_passkey_display
// passkey is generated by the stack and provided in callback for display to the user
static void pairing_passkey_display(struct bt_conn *conn, unsigned int passkey)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	printk("Passkey for %s: %06u\n", addr, passkey);
}

// TODO implement auth_confirm
static void auth_confirm(struct bt_conn *conn, unsigned int passkey)
{

	default_conn = conn;
	char addr[BT_ADDR_LE_STR_LEN];
	char passkey_str[7];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	snprintk(passkey_str, 7, "%06u", passkey);
	printk("\nConfirm passkey for %s: %s\n\n", addr, passkey_str);
	printk("Press button A for YES or button B for NO\n");
	authenticating = true;
}

// TODO implement auth_cancel
static void auth_cancel(struct bt_conn *conn)
{
	printk("Pairing cancelled\n");
	authenticating = false;
}
```

pairing_passkey_display is called by the stack with a system-generated six digit passkey. It is up to you what you do with this random number. In this case, it is simply outputting it to the console.

auth_confirm is called when the user should be invited to confirm that the passkey displayed on the console is the same as the value displayed on the screen of the other device.

auth_cancel will be called if pairing is cancelled or if the user indicates that the number displayed on the serial console is not the same as the number displayed on the smartphone.

Test

Build your code and flash the binary to your board. Clear any previous pairing with this board on your smartphone.

Connect your board to your computer using a USB cable. Determine which USB port number has been assigned to it. On Windows, this information is to be found within Device Manager.

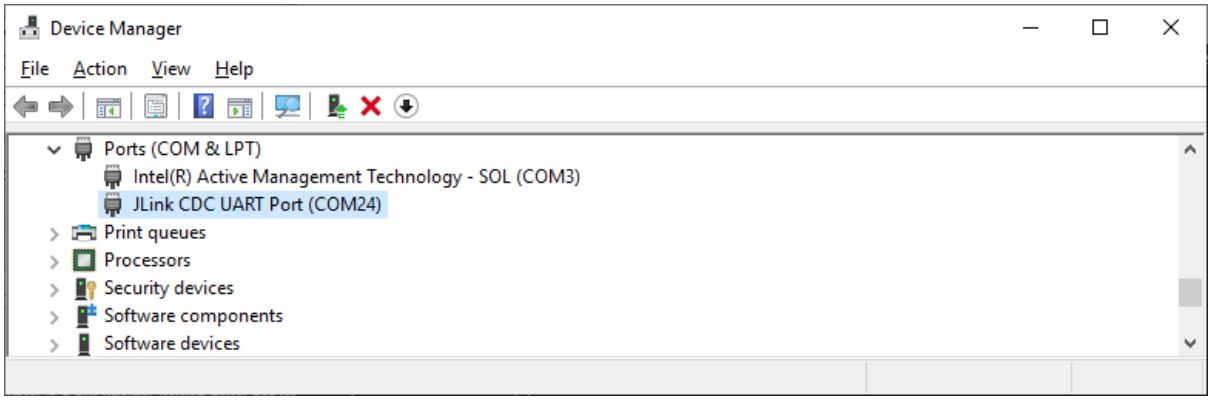


Figure 26 – Serial terminal USB port

Open a serial terminal program, such as Putty, and connect to your board.

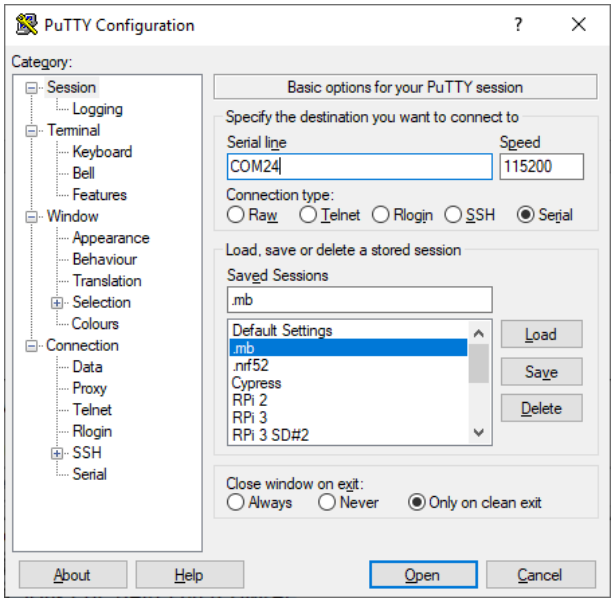


Figure 27 – Connecting to the serial console with Putty

Open the nRF Connect application, discover and then connect to your board, which will be advertising with the name NCLEC. On connecting, you should be immediately prompted to pair with the device. Confirm that pairing should proceed by pressing the appropriate button on your smartphone.

LE Secure Connections pairing with the Numeric Comparison association model should be used, as described next.

You should then see a six-digit numeric code displayed on both the smartphone and in the terminal connected to your board. The board will prompt you to confirm that you are seeing the same numeric code displayed on both devices. If the numeric code is the same on both devices, press whichever button you have programmed to act as button A You will then see a message displayed on the terminal which indicates that security level 4 has been engaged.

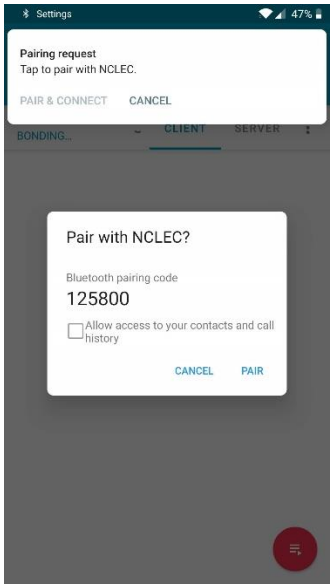


Figure 28 – Numeric Comparison authentication on an Android smartphone

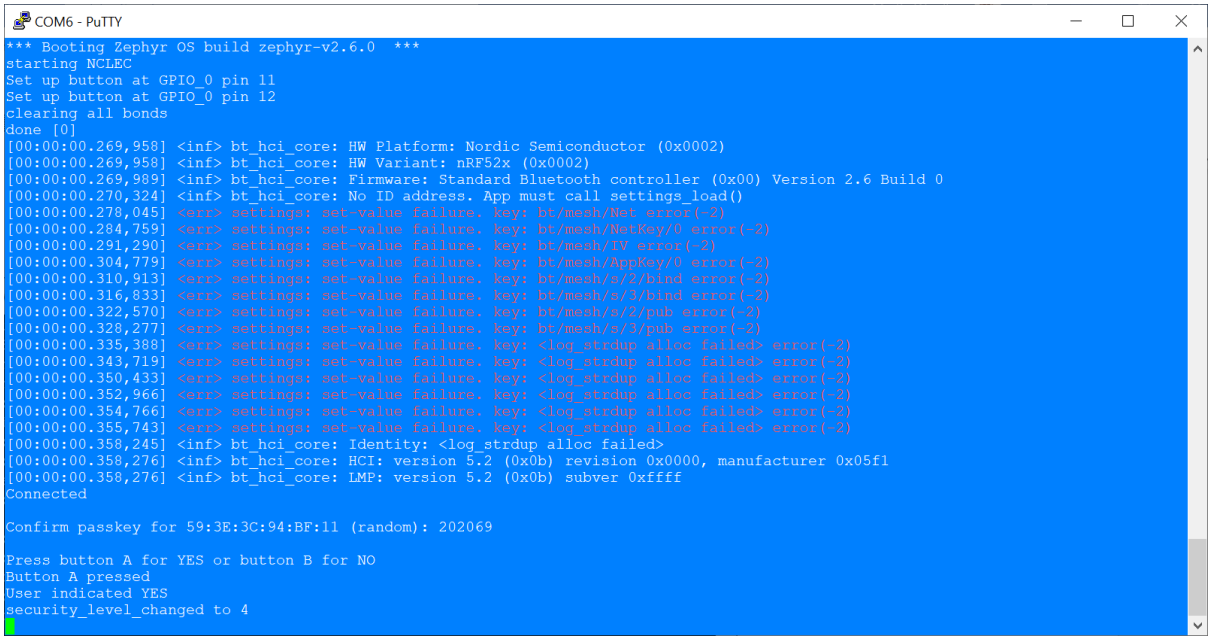


Figure 29 – Numeric comparison authentication in the peripheral's console

If you have a sniffer, capture a trace while pairing. You should see an exchange of messages similar to that shown in Figure 30.

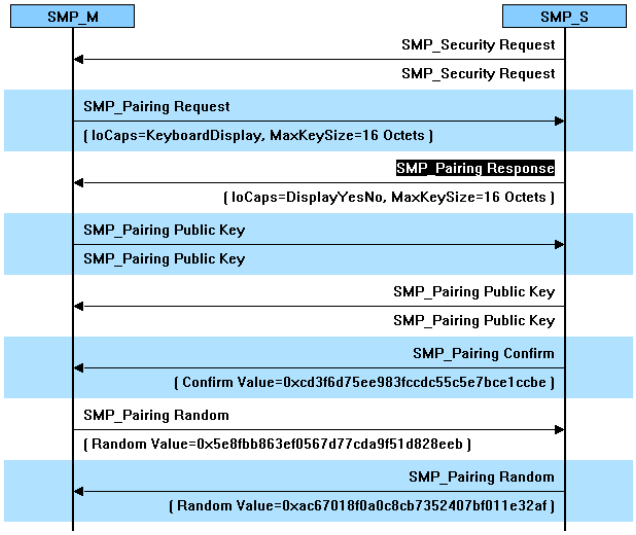


Figure 30 – Trace: LE Secure Connections pairing

If you examine the Pairing Request and Pairing Response messages, the IO capabilities and other pairing parameters should reveal how LE Secure Connections with Numeric Comparison was selected. See Figure 31.

Data exchanged after pairing, whether in the same session or after disconnecting and reconnecting, will be encrypted at the link layer. Your sniffer will not be able to show you this data.

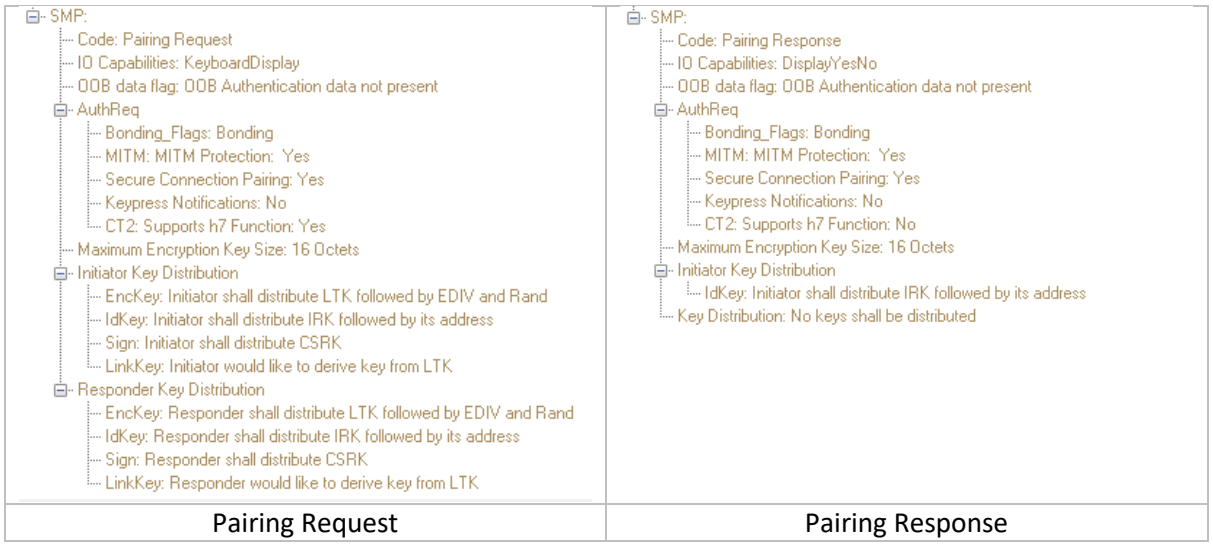


Figure 31 – Pairing feature request and response details

4.4.3 LE Legacy Pairing

This section examines LE Legacy Pairing with the Just Works association model.

Code

Create a Zephyr project called blessg_ex3. Copy the content of this resource's code\start_points\blessg_ex3 folder into your blessg_ex3 and create a sub-folder called build. All of the required code, except for that related to security, is already included in this source.

Your task now is to complete those sections marked with TODO comments so that your board implements security mode 1, level 2 and indicates IO capabilities and other pairing preferences that

should result in LE Legacy pairing with the Just Works association model being used when you pair with your smartphone.

The changes to be made are similar to those made in exercise 2 and fall into three areas:

1. Request security mode 1, level 2
2. Register callback functions relating to pairing and security level changes
3. Implement the callback function

Proceed by adding the code fragments below in the appropriate places within the main.c source file, pausing to review the code and appreciate what it achieves.

Step 1 - Request Security Mode 1, level 2

Update the *connected* function so it includes the call to *bt_conn_security* shown here:

```
static void connected(struct bt_conn *conn, u8_t err)
{
        if (err)
        {
        }
        else
        {
                printk("Connected\n");
                default_conn = bt_conn_ref(c
                // TO DO request mode 1 level 4 security
                int rc = bt_conn_security(default_conn, BT_SECURITY_L2);
        }
}
```

Zephyr uses the constant BT_SECURITY_L2 to indicate that LE Legacy pairing with no authentication should be used.

Step 2 - Register Callback Functions Relating to Pairing and Security-Level Changes

Declare a struct of type bt_conn_auth_cb. This struct is used to register a callback function that will be involved in implementing pairing.

```
// TODO specify callbacks for pairing - IO Capabilities are implied from this struct
static struct bt_conn_auth_cb pairing_cb_display = {
                .passkey_display = NULL,
                .passkey_entry = NULL,
                .cancel = pairing_cancel,
};
```

Note that the passkey-related struct properties are NULL. This is because Just Works pairing does not involve the display or entry of a passkey value.

Now, modify the main function to register the three callback functions.

```
void main(void)
{

        printk("starting JW\n");
        int err;
        configureButtons();
        err = bt_enable(bt_ready);
        if (err)
        {
                return;
        }
```

```
        clear_all_bonds();

        bt_conn_cb_register(&conn_callbacks);

    // TODO register callback for authentication step
        bt_conn_auth_cb_register(&pairing_cb_display);
}
```

Update the conn_callbacks struct so that it includes a security_changed callback function. This function will be called when the security level has been changed to level 2 after pairing.

```
// TODO specify callback for security level changing
static struct bt_conn_cb conn_callbacks = {
                .connected = connected,
                .disconnected = disconnected,
                .security_changed = security_level_changed,
};
```

Step 3 - Implement the Callback Functions

Add the following callback function. It will simply write a message to the console to indicate when the security level has been changed and to what level. This will confirm we've switched to level 2.

```
// TODO implement security_level_changed
static void security_level_changed(struct bt_conn *conn, bt_security_t level, enum
bt_security_err err)
{
        printk("security_level_changed to %d\n", level);
}
```

Finally, add the following function to implement the authentication cancellation callback function.

```
// TODO implement auth_cancel
static void pairing_cancel(struct bt_conn *conn)
{
        char addr[BT_ADDR_LE_STR_LEN];
        bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
        printk("Pairing cancelled by device: %s\n", addr);
}
```

auth_cancel will be called if pairing is cancelled by the user.

Test

Build your code and flash the binary to your board. Clear any previous pairing with this board on your smartphone.

Connect your board to your computer using a USB cable. Determine which USB port number has been assigned to it.

Open the nRF Connect application, discover, and then connect to your board, which will be advertising with the name JW. Pair with the board, either from your smartphone's Bluetooth settings screen or by using nRF Connect to discover, and then connect to the board. Aside from confirming that pairing should proceed, you should not need to interact with your smartphone or the peripheral during pairing.

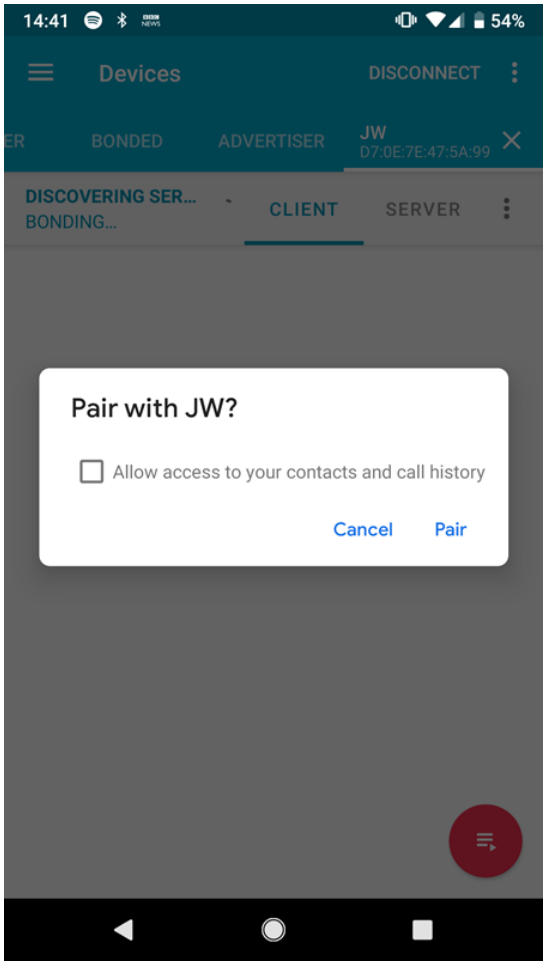


Figure 32 – Confirming that pairing should be performed

The console should contain something like this:

```
*** Booting Zephyr OS build zephyr-v2.6.0  ***
starting JW
About to initialise custom service
done initialise custom service
About to load settings
done loading settings
About to start advertising
done started advertising
Advertising successfully started
clearing all bonds
done [0]
[00:00:00.396,728] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.396,759] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
[00:00:00.396,759] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00)
Version 2.6 Build 0
[00:00:00.396,972] <wrn> bt_ecc: ECC HCI commands not available
[00:00:00.397,003] <inf> bt_hci_core: No ID address. App must call settings_load()
[00:00:00.405,151] <inf> bt_hci_core: Identity: C9:D7:58:2D:C7:55 (random)
[00:00:00.405,181] <inf> bt_hci_core: HCI: version 5.2 (0x0b) revision 0x0000, manufacturer
0x05f1
[00:00:00.405,181] <inf> bt_hci_core: LMP: version 5.2 (0x0b) subver 0xffff
Connected
security_level_changed to 1
Disconnected (reason 19)
```

Figure 33 – Just Works pairing in the peripheral's console

If you have a sniffer, capture the communication between the two devices during pairing. You should see SMP messages like these:

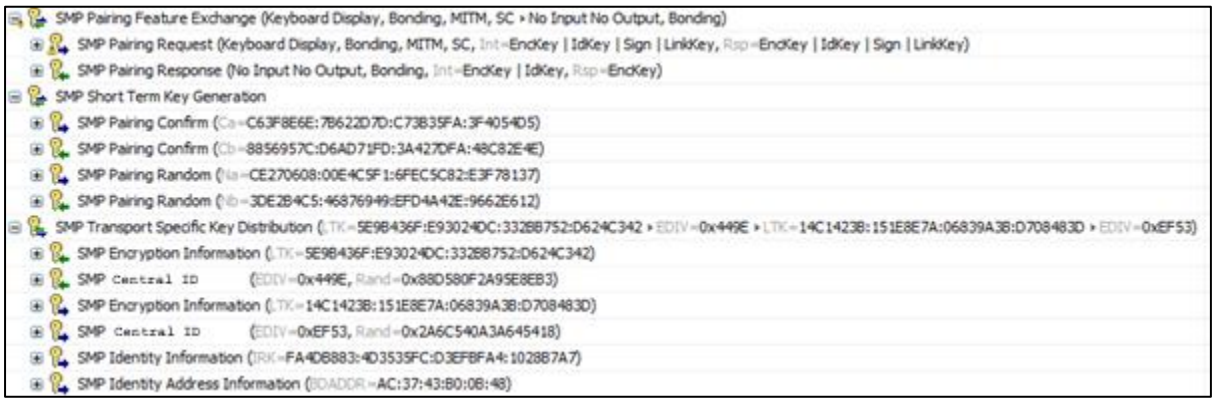


Figure 34 – Trace: LE Legacy Pairing with Just Works

Use nRF Connect to discover and connect to your board. Write a value of 0x03 to the writable characteristic and then read it back into the other characteristic. If you have captured the communication using a sniffer, you will see something like this:

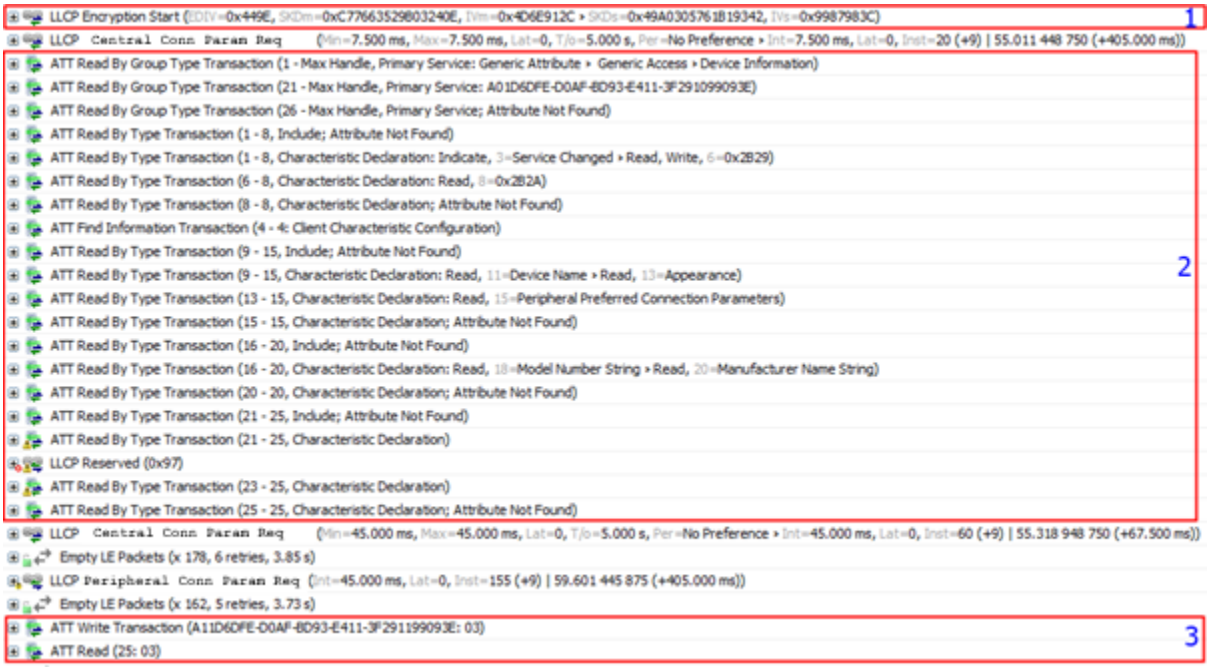


Figure 35 – Trace: starting encryption, service discovery, and then writing and reading characteristics

You should note the following points:

1. Encryption has been started on the link, as required.

2. The *ATT Read by Group Type, ATT Read by Type and ATT Find Information* PDUs seen in the trace above constitute service discovery taking place. Bluetooth Core Specification version 5.2 Vol 3 Part G 8.1 says "*The list of services and characteristics that a device supports is not considered private or confidential information, and therefore the Service and Characteristic Discovery procedures shall always be permitted*".

3. The sniffer is able to show us the content of all PDUs captured, including the values written to and read from our characteristics, despite the fact that the link is encrypted. LE Legacy Pairing using Just Works or Passkey Entry uses a very weak Temporary Key (TK) during pairing, whose value the sniffer was able to establish by using a brute force attack. Once the TK is established, the remainder of the pairing process and everything that follows is rendered insecure. As the Bluetooth Core Specification states: "*In LE Legacy Pairing ….. Just Works and Passkey Entry do not provide any passive eavesdropping Protection"* – see Vol 1 Part A section 5.4.1.

4.4.4 GATT Attributes and Permissions

Attributes in the *attribute table* have a series of permissions associated with them. As described elsewhere in this document, four types of permission exist: *access permissions*, *encryption permissions*, *authentication permissions,* and *authorization permissions*.

In this exercise, you will associate encryption permissions with the *output characteristic* and both encryption and authentication permissions with the *input characteristic*. This should mean that you may read from the output characteristic if the link is encrypted and regardless of how pairing was performed, whereas you will only be able to write to the input characteristic if the link is encrypted and the devices have been paired using an authenticating association model.

Code - Part 1

Create a Zephyr project called blessg_ex4. Copy the content of this resource's code\start_points\blessg_ex4 folder into your blessg_ex4 and create a sub-folder called build. All of the required code except for that related to attribute security is already included in this source.

Your task now is to change the attribute permissions assigned to the two characteristics in the custom service.

Modify Attribute Permissions

Proceed by modifying the characteristic permissions in the custom_service.c file, as shown here:

```
static struct bt_gatt_attr attrs[] = {
                BT_GATT_PRIMARY_SERVICE(BT_UUID_CUSTOM_SERVICE),
                BT_GATT_CHARACTERISTIC(BT_UUID_INPUT, BT_GATT_CHRC_WRITE,
                                       BT_GATT_PERM_WRITE_AUTHEN, NULL, write_input, NULL),
                BT_GATT_CHARACTERISTIC(BT_UUID_OUTPUT, BT_GATT_CHRC_READ,
                                       BT_GATT_PERM_READ_ENCRYPT, read_output, NULL, NULL),
};
```

Test - Part 1

Build and install this code onto your board. Clear any previous pairing with this board on your smartphone.

Launch the nRF Connect application on your smartphone. Scan for and then connect to your board, which should be advertising with the name JWLECATT.

Expand the service entry so that both characteristics are visible in the UI. Attempt to read the output characteristic (which supports the READ operation). Due to the encryption permission attached to this characteristic, pairing will be triggered, and due to the I/O capabilities implied by the bt_conn_auth_cb struct in the code, LE Secure Connections pairing with Just Works will be attempted.

```
static struct bt_conn_auth_cb pairing_cb_display = {
                .passkey_display = NULL,
```

```
                .passkey_entry = NULL,
                .cancel = pairing_cancel,
};
```

But there's a problem, and your smartphone should disconnect. Swipe right in the nRF Connect UI and you will see the application's log. This indicates that there has been a GATT AUTH FAIL.

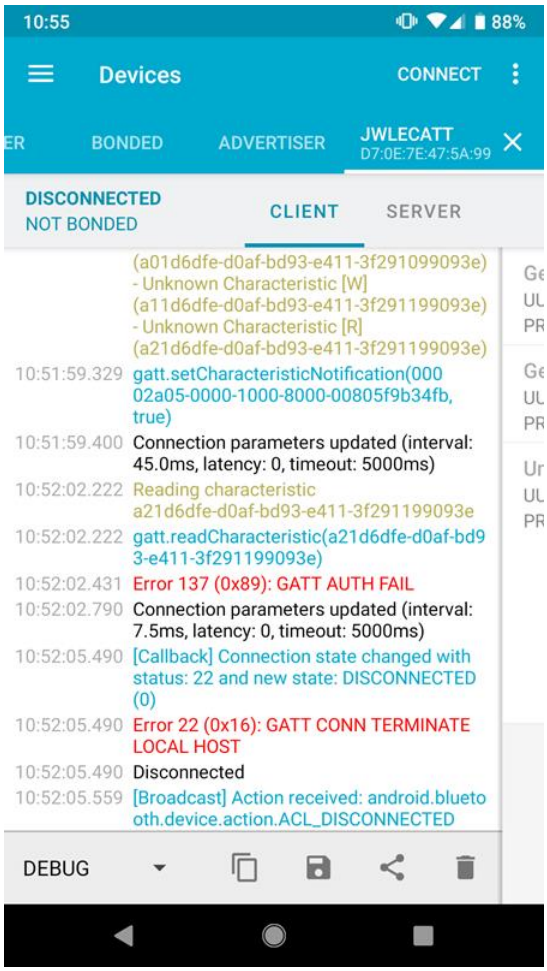


Figure 36 – Authentication requirements failure in nRF Connect

Open prj.conf in your editor and note the inclusion of CONFIG_BT_SMP_SC_ONLY=y. This is the reason for the failure. The definition of *Secure Connections Only Mode* stipulates that authenticated pairing must have been used and Just Works does not include authentication.

A trace of the communication between devices confirms this.

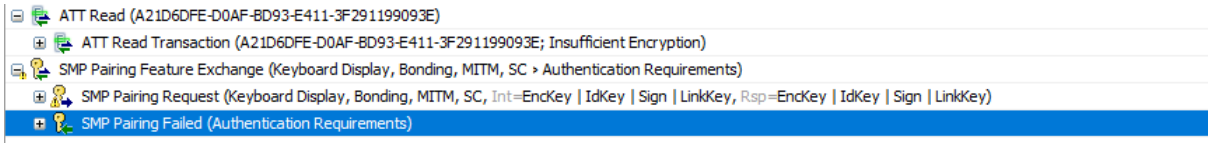


Figure 37 – Trace: Authentication requirements failure

So, the observed behaviour is correct, given that Secure Connections Only mode has been requested in our build configuration properties.

Remove CONFIG_BT_SMP_SC_ONLY=y from prj.conf, build and install again. Connect to the device using nRF Connect. Once again, expand the service entry so that both characteristics are visible in the UI. Attempt to read the output characteristic (which supports the READ operation). Due to the encryption permission attached to this characteristic, pairing will again be triggered. This time, you will see the security level changing to 2, in the peripheral's attached console, as encryption is enabled.

```
*** Booting Zephyr OS build zephyr-v2.6.0  ***
starting JWLECATT
Set up button at GPIO_0 pin 11
Set up button at GPIO_0 pin 12
clearing all bonds
done [0]
[00:00:00.399,291] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.399,291] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
[00:00:00.399,291] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00)
Version 2.6 Build 0
[00:00:00.399,627] <inf> bt_hci_core: No ID address. App must call settings_load()
[00:00:00.399,993] <inf> bt_hci_core: Identity: C9:D7:58:2D:C7:55 (random)
[00:00:00.400,024] <inf> bt_hci_core: HCI: version 5.2 (0x0b) revision 0x0000, manufacturer
0x05f1
[00:00:00.400,024] <inf> bt_hci_core: LMP: version 5.2 (0x0b) subver 0xffff
Connected
security_level_changed to 2
read_output
```

The peripheral code has initialized the output characteristic's value to 64, and, since the link is now encrypted, you should see this value displayed in nRF Connect as hex value 0x40.

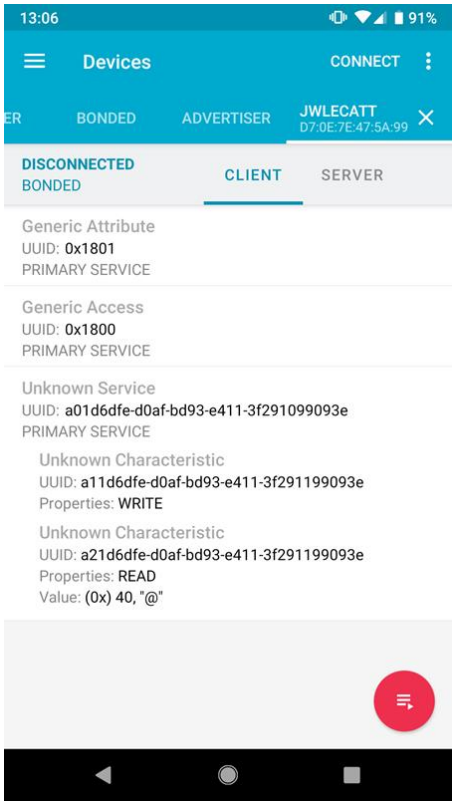


Figure 38 – nRF Connect with successfully read characteristic value

Next, attempt to write a value of 0x05 to the input characteristic (which supports the WRITE operation). The operation will not succeed. Read the output characteristic. It's value will not have changed.

Pairing was accomplished using LE Secure Connections (assuming your smartphone supports it). So, if you are running a sniffer, it will not have been able to crack the encryption keys used and, therefore, not be able to decrypt the part of your testing session that involves the read and write GATT procedures. The behaviour of the client application was as expected, however, so you can conclude that things are working as required.

If you were to try the same test but with LE Legacy Pairing and Just Works having been used, your sniffer could probably decrypt the communication and you would see something like this:

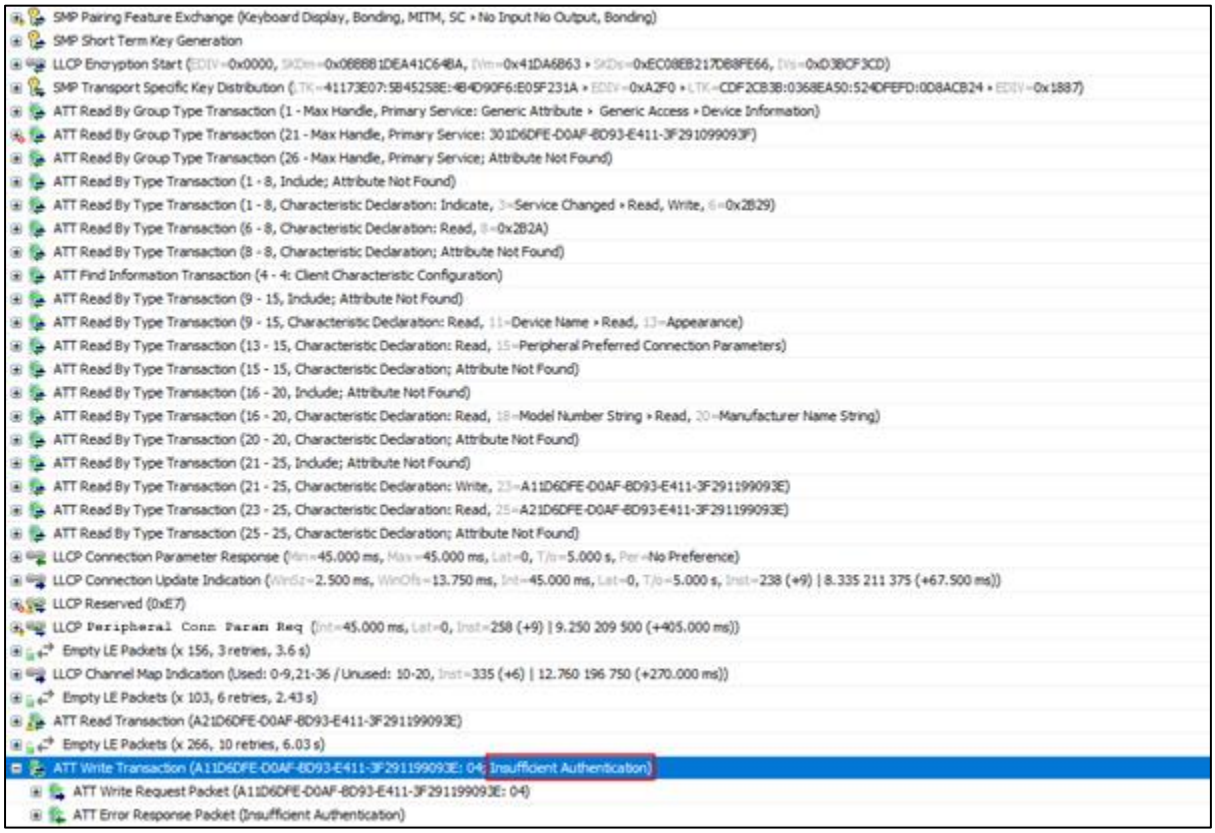


Figure 39 – Trace: Insufficient authentication

Code - Part 2

Your next task is to modify the behaviour of your board so that Numeric Comparison is used when pairing. This is an authenticated form of pairing and should therefore satisfy the requirements of the authentication permission attached to the input characteristic.

Create a Zephyr project called blessg_ex5. Copy the content of this resource's code\start_points\blessg_ex5 folder into your blessg_ex5 and create a sub-folder called build. All of the required code is already included in this source and the prj.conf build configuration file includes CONFIG_BT_SMP_SC_ONLY=y, indicating that Secure Connections Only mode is required. Note the value of the *pairing_cb_display* struct and associated callback functions. This code, together with the switch to security mode 1, level 4, ensures LE Secure Connections with Numeric Comparison is used.

```
// will result in Numeric Comparison being used
static struct bt_conn_auth_cb pairing_cb_display = {
                .passkey_display = pairing_passkey_display,
                .passkey_confirm = auth_confirm,
                .cancel = pairing_cancel,
};

static void connected(struct bt_conn *conn, u8_t err)
{
        if (err)
        {
        }
        else
        {
                printk("Connected\n");
                default_conn = bt_conn_ref(conn);

                int rc = bt_conn_security(default_conn, BT_SECURITY_L4);
        }
}
```

Test - Part 2

Build and install this code onto your board. Clear any previous pairing with this board on your smartphone. Perform the same tests as were performed in Testing Part 1. This time, the write operation should succeed, and whatever value you write to the input characteristic, should be retrieved and displayed when you read from the output characteristic. Remember that values written to the input characteristic must be between 0 and 10 (decimal).

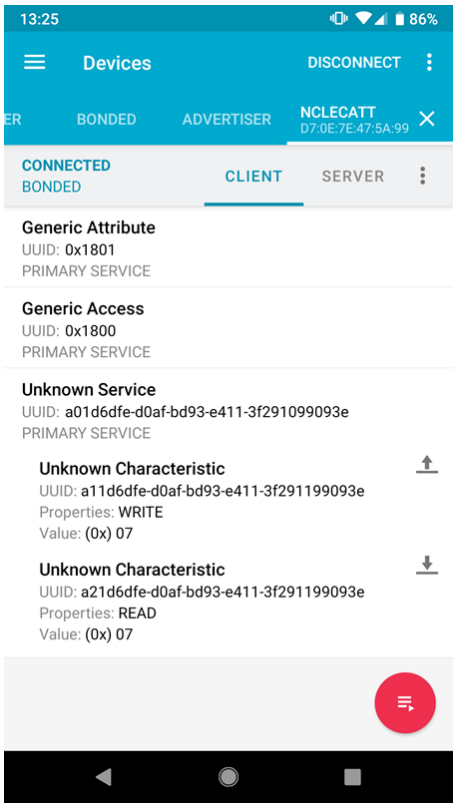


Figure 40 – nRF Connect showing successfully written to and read from characteristics

Code - Part 3

The next task involves authorization permissions. You will implement code that only allows the input characteristic to be written to if the two connected devices are very close together. Signal strength (RSSI) is used to assess the proximity of the two devices. If the RSSI is less than -30 dBm then the write request will be denied, and a response of Insufficient Authorization will be returned.

Create a Zephyr project called blessg_ex6. Copy the content of this resource's code\start_points\blessg_ex6 folder into your blessg_ex6 and create a sub-folder called build.

Open services\custom_service.c in your editor. With the Zephyr SDK, encryption and authentication permissions are indicated using constants provided to the BT_GATT_CHARACTERISTIC macro. Authorization permissions do not use the same approach in code. All that is required is to implement the authorization logic in the function that handles the relevant operation on the characteristic and either allow it to proceed or return an Insufficient Authorization error in the ATT response.

Find the //TODO comment in the write_input function. After the comment, add this code:

```
        printk("Checking RSSI\n");
        s8_t rssi = get_rssi();
        printk("RSSI: %01hhd \n",rssi);

        if (rssi < -30) {
                printk("devices not close enough - requested operation not authorized\n");
                return BT_GATT_ERR(BT_ATT_ERR_AUTHORIZATION);
        }
```

The *get_rssi* function is implemented in utils/rssi.c. Also, in that source file is a function called *read_rssi*. This uses a blocking function call to send an HCI command, *hci_read_rssi* to the Bluetooth controller. The returned value is stored in a variable. Take a look in main.c and you will note that *read_rssi* is called once a second by a Zephyr delayed timer. Consequently, our authorization code is examining the most recently measured RSSI value, which should be, at most, one second old and using this as the basis for the authorization decision. You may find it useful or interesting to experiment with different RSSI limits in the authorization check.

Test

Build and install this code onto your board. Clear any previous pairing with this board on your smartphone.

Using nRF Connect, discover and connect to the board, which should be advertising with the name NCLECAUT. When prompted to, pair with the board, observing the serial console connected to your board to see the passkey.

With your smartphone a couple of meters away from the board, attempt to write a value of 0x04 to the input characteristic. The attempt should fail. The console will indicate this with a message. Now, place your phone right next to the board and try again. This time the write should be allowed to succeed.

```
Connected
starting RSSI sampling
[00:00:06.479,583] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -20 dB.
[00:00:06.929,199] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -24 dB.
RSSI: -22
[00:00:07.964,172] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -36 dB.
RSSI: -45
RSSI: -39
[00:00:09.494,140] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -39 dB.
RSSI: -43
[00:00:10.259,124] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -53 dB.
RSSI: -49
RSSI: -57
read_output
[00:00:12.644,104] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -66 dB.
RSSI: -67
RSSI: -57
[00:00:15.029,052] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -55 dB.
RSSI: -49
RSSI: -52
RSSI: -54
Checking RSSI
RSSI: -54
devices not close enough - requested operation not authorized
RSSI: -58
RSSI: -58
RSSI: -49
RSSI: -46
RSSI: -52
RSSI: -58
RSSI: -54
[00:00:25.153,869] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -28 dB.
RSSI: -22
RSSI: -23
[00:00:26.323,791] <inf> bt_ctlr_hci: handle: 0x0000, rssi: -21 dB.
RSSI: -23
Checking RSSI
RSSI: -23
write_input [5]
RSSI: -22
RSSI: -21
RSSI: -22
```

Your board and smartphone were paired using LE Secure Connections and so tracing their interaction is not possible. With pairing temporarily disabled in the Zephyr code, the following trace was captured and clearly shows the ATT Insufficient Authorization response.

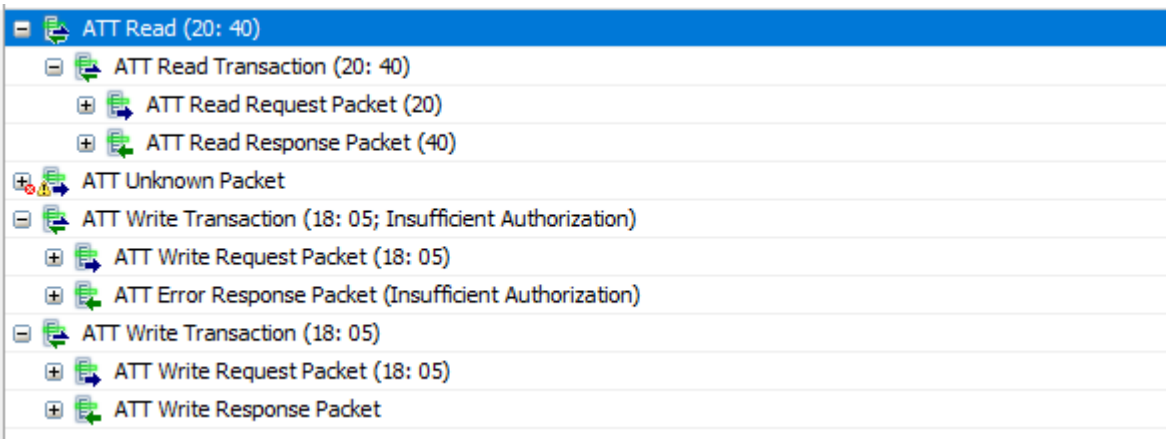


Figure 41 - Trace: Insufficient Authorization

4.4.5 Privacy

Code – Part 1

The next task will involve the use of private resolvable addresses.

Create a Zephyr project called blessg_ex7. Copy the content of this resource's code\start_points\blessg_ex7 folder into your blessg_ex7 and create a sub-folder called build.

Test – Part 1

Build and install this code onto your board. Clear any previous pairing with this board on your smartphone. Open the nRF Connect application on a smartphone. Observe the address advertised by the Zephyr device, which will be using the name PK_LEC_PRIV.

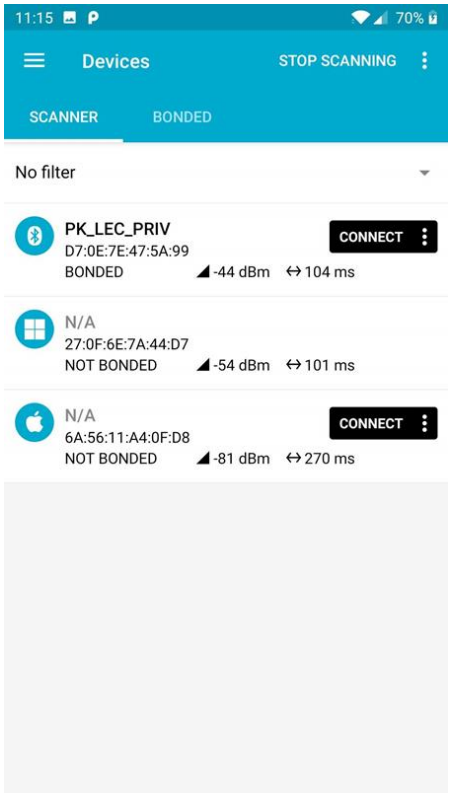


Figure 42 – nRF Connect showing static address

In the example shown above, the address is D7:0E:7E:47:5A:99. The most significant byte is 0xD7, which is binary 11010111. Per Bluetooth Core Specification Vol 6 Part B section 1.3.2.1, given the two most significant bits are both 1, this is a static address.

Refresh the scanner screen in nRF Connect periodically and keep doing so for at least 15 minutes. The advertised address will not change.

This test has confirmed that private addresses are not currently in use.

Code – Part 2

Enabling resolvable private address generation in Zephyr is performed by adding some build configuration properties to the prj.conf file. Open this file in your editor and locate the TODO comment.

Add the following properties after the comment:

```
CONFIG_BT_PRIVACY=y
CONFIG_BT_CTLR_PRIVACY=y
CONFIG_BT_RPA_TIMEOUT=60
```

Test – Part 2

Build and install the modified code onto your board. Clear any previous pairing with this board on your smartphone. Open the nRF Connect application on a smartphone. Connect to the device advertising with the name PK_LEC_PRIV and, while monitoring the serial console, pair. You should be required to enter a passkey, which is displayed in the console into your smartphone.

```
*** Booting Zephyr OS build zephyr-v2.6.0  ***
starting PK_LEC_PRIV
settings_load result: 0
[00:00:00.412,109] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.412,109] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
[00:00:00.412,139] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00)
Version 2.6 Build 0
[00:00:00.412,445] <inf> bt_hci_core: No ID address. App must call settings_load()
[00:00:00.412,841] <inf> bt_hci_core: Identity: C9:D7:58:2D:C7:55 (random)
[00:00:00.412,841] <inf> bt_hci_core: HCI: version 5.2 (0x0b) revision 0x0000, manufacturer
0x05f1
[00:00:00.412,872] <inf> bt_hci_core: LMP: version 5.2 (0x0b) subver 0xffff
*** Booting Zephyr OS build zephyr-v2.6.0  ***
starting PK_LEC_PRIV
settings_load result: 0
[00:00:00.260,009] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.260,009] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
[00:00:00.260,009] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00)
Version 2.6 Build 0
[00:00:00.260,345] <inf> bt_hci_core: No ID address. App must call settings_load()
[00:00:00.260,925] <inf> bt_hci_core: Identity: C9:D7:58:2D:C7:55 (random)
[00:00:00.260,925] <inf> bt_hci_core: HCI: version 5.2 (0x0b) revision 0x0000, manufacturer
0x05f1
[00:00:00.260,925] <inf> bt_hci_core: LMP: version 5.2 (0x0b) subver 0xffff
Connected
Passkey for 56:0A:AE:9D:F4:0C (random): 489845
security_level_changed to 4
```

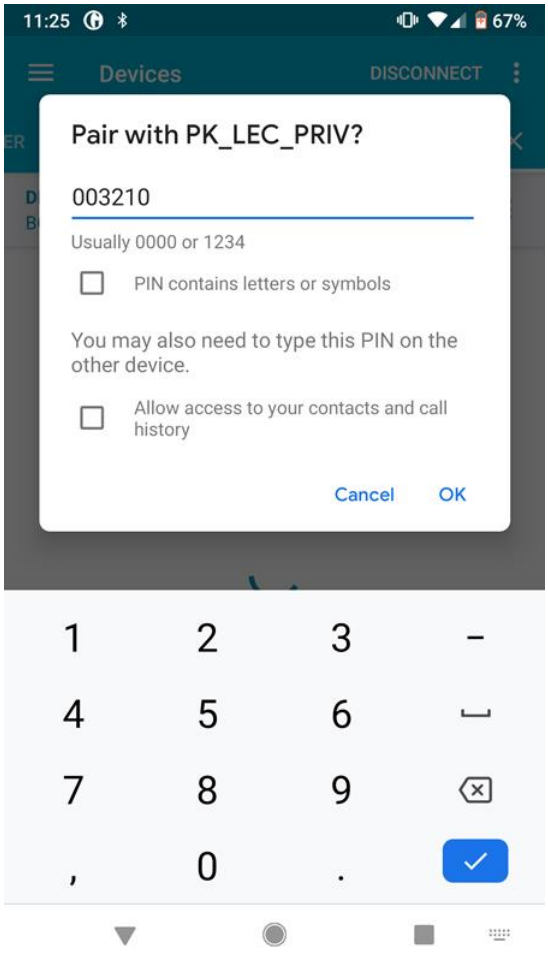


Figure 43 – Passkey Pairing on Android

Disconnect nRF Connect from your Zephyr board. It should resume advertising. nRF Connect will show that the two devices are bonded and continue to show the original static address as this device's address. It is able to do this because, having paired, it has an IRK with which to perform address resolution.

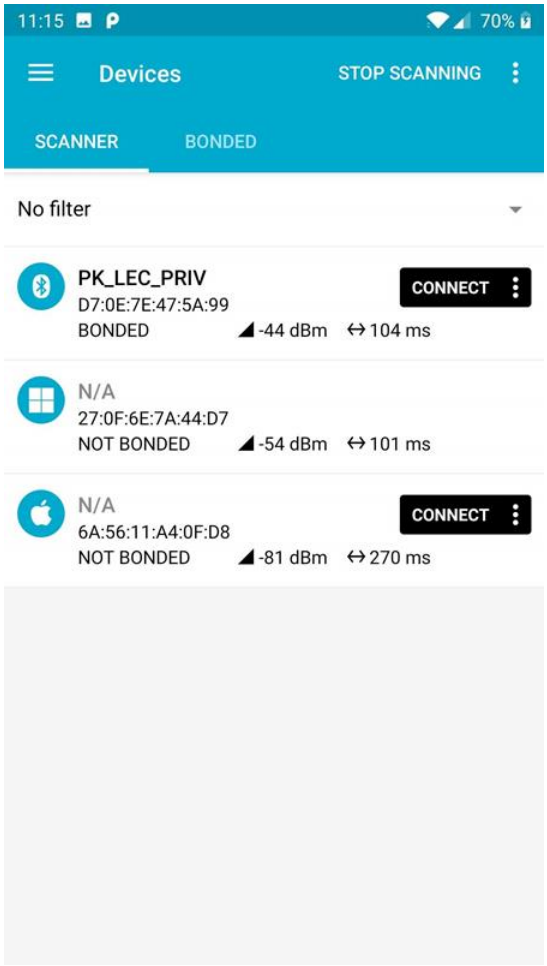


Figure 44 – nRF Connect on a bonded smartphone, still showing static address

Either, forget the pairing on your smartphone or, using another smartphone that you have not paired with the Zephyr board, run nRF Connect again. You should now see a resolvable private address attributed to the PK_LEC_PRIV device. This is because this smartphone does not have the ability to resolve the private address.

Keep refreshing the nRF Connect scanner screen for a few minutes. The address shown for the Zephyr test device will change every minute (due to the 60-second value assigned to the CONFIG_BT_RPA_TIMEOUT property in prj.conf). If you are testing with two different smartphones, perform the scanner screen refresh at the same time on both devices. You will find that the address shown on the bonded device does not change, whereas it does on the unbonded smartphone.

This test has confirmed that resolvable private addresses are in use and that a bonded device is able to resolve the address.

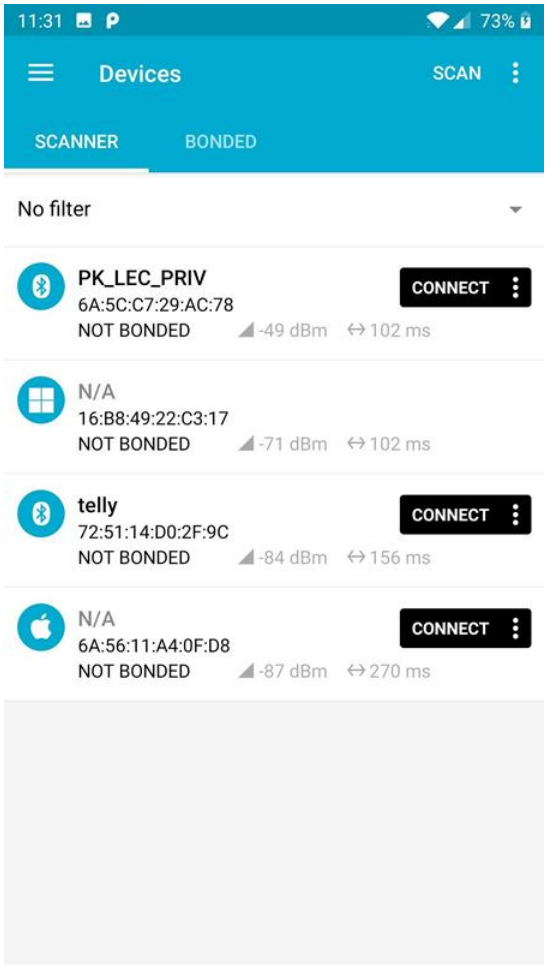


Figure 45 – nRF Connect showing a resolvable private address

Note that the address shown (6A:5C:C7:29:AC:78) has a most significant byte of 0x6A, which is binary 01101010. Per Bluetooth Core Specification Vol 6 Part B section 1.3.2.2, given the most significant bit is 0 and the next most significant bit is 1, this is a resolvable private address.

4.5 Security and Implementation Issues

Origins of Security Issues

In simple terms, security issues can be said to arise from three fundamentally different sets of root causes.

1. Specification Errors

If a security issue arises from the way a Bluetooth feature is defined in the applicable Bluetooth specification(s), then the root cause is the specification itself and it is valid to describe the issue as a *Bluetooth security issue*. Issues of this type should be reported to the Bluetooth Special Interest Group for evaluation. If warranted, specification corrections that address the reported issue will be published.

2. Implementation Errors

An implementation may be functionally correct and able to pass all the relevant Bluetooth qualification tests but still contain security vulnerabilities due to technical coding errors and issues that do not affect the observable behaviour of the product with respect to its use of Bluetooth under test conditions. Issues of this sort are *technical quality issues*.

3. Poor Implementation Choices

Implementors may make poor choices regarding the security features to be supported by a product but implement them quite correctly. This might be a result of security requirements not having been properly assessed for the product. A poor choice is a poor choice, regardless of how well it is implemented.

Specification Compliance

The Bluetooth Core Specification describes and defines the security features of a Bluetooth stack. Bluetooth profiles define the requirements for using those features in the context of a given set of use cases or product type. Like all specifications, they are just specifications, and it is incumbent on developers and other product development professionals to design and implement products correctly and in accordance with the applicable specifications.

Bluetooth test cases seek to verify the correct implementation of and/or use of Bluetooth security features. The testing method used is called *conformance testing "defined in ISO/ IEC 9646-1 and ISO/IEC 9646-2, both part of the OSI Conformance Testing Methodology and Framework (CTMF)"*[1]. A high-level overview of conformance testing together with a collection of applicable references is available in Wikipedia: https://en.wikipedia.org/wiki/Conformance_testing.

A device exhibiting each of the expected outcomes defined for each of the associated test cases should be regarded as having demonstrated correct behaviors, as required for specification compliance and interoperability to be achieved. This should not be regarded as absolute assurance that the product is secure, however.

[1] Security Manager Protocol Bluetooth Test Suite revision SM.TS.5.1.0

Technical Quality Issues

Writing secure code takes skill, experience, and an understanding of best practice.

There are various ways poorly written code can introduce security vulnerabilities to a product. Common issues include a failure to check and enforce buffer length restrictions that can make buffer or stack overflow attacks possible and the inadvertent leaking of sensitive information.

Bluetooth test procedures cannot assess the internal quality of device code. It is, therefore, for the developer and his/her company's development and quality assurance practices to ensure that the technical quality of device code is fit for purpose and does not introduce security vulnerabilities.

5. Conclusions

Bluetooth LE includes a range of security featuresthat can be used to meet requirements relating to confidentiality, authentication, privacy, and access to device data. Used properly and with a properly established and articulated understanding of product security requirements, secure Bluetooth products can be created. This is achieved through a combination of making the right choices of security features from the Bluetooth Core Specification, proper compliance with all applicable specifications, thorough testing (which includes testing of device security), and following best practice when it comes to writing code.

Hopefully, this study guide helped you gain a better understanding of security in general as well as security as it applies to Bluetooth LE connectable devices. Do not forget, this is not a substitute for the Bluetooth Core Specification, the Bluetooth GATT profiles, and the Bluetooth test suites.

Good luck with your next product development project.