Bonus Web Chapter 2

Bluetooth is a wireless protocol released in 1998 with the goal of providing wireless connectivity for a variety of devices. The most commonly envisioned scenario was cell-phone syncing with the (now ubiquitous) Bluetooth headset. Since then, Bluetooth has gained traction as a general-purpose, short-range wireless cable replacement protocol. Its use has been expanded into some predictable markets (such as video game controllers) as well as into some fields that the original specification writers probably never guessed, such as medical devices.

Bluetooth is a markedly different protocol than 802.11.Nearly all commercial products that implement 802.11 have a 32- or 16-bit CPU on board. Many Bluetooth devices, however, are constrained in terms of battery power and CPU cycles. In short, controlling an 802.11 chipset, even one that is implemented largely in hardware, requires a very complex piece of software to be executed on an external CPU (the driver and associated userland code). In contrast, Bluetooth chipsets can be controlled via a simple serial interface, and potentially driven by an 8-bit microcontroller. Some Bluetooth chipsets basically drive themselves.

The core Bluetooth protocol is currently over 1700 pages, and that's not including any of the profiles (which are like application-level protocols). The following brief overview is designed to spare you and your machine the trouble of reading/loading that 1700-page file. Of course, the standard is the final word, but our hope is that this brief overview will be much easier on your eyes.

The problem every author faces when trying to describe Bluetooth is that the term *Bluetooth* is as technically precise as *Internet*. The Internet is composed of a large stack of independent protocols, most of them intentionally independent from the layers above and below it. Bluetooth, on the other hand, encompasses all the layers up to the application level. These layers are not nearly as well separated, making any explanation via comparison to more familiar protocols inherently fuzzy. This Bonus Web Chapter attempts to explain the most commonly used protocols in the Bluetooth suite in enough detail that, for those who are not yet familiar with Bluetooth, the chapters on attacking Bluetooth security can be understood without an outside reference.

High-Level Review

The goal of the following section is to describe the interactions of Bluetooth devices at a high level and without assuming significant knowledge of the underlying protocols. Basic concepts such as device discovery, frequency hopping, and piconets will be covered. Details on the protocol stack and implementation will come later.

The Bluetooth specification defines 79 channels across the 2.4-GHz ISM band, each channel occupying 1-MHz of spectrum. Devices hop across these channels at an impressive rate of 1600 times a second (every 625 microseconds). Whenever you find yourself wondering why everything in Bluetooth seems so convoluted, keep in mind that devices are constantly moving across these channels.

This channel-hopping technique is known as *Frequency Hopping Spread Spectrum (FHSS),* and in current Bluetooth implementations, the user can achieve a rate of 3 Mbps

of bandwidth across 100 meters. FHSS provides robustness against noisy channels by rapidly changing frequencies. Later revisions of the standard have added support for adaptive hopping, which allows noisy channels to be detected and avoided all together.

Any set of devices that wish to communicate using Bluetooth need to be on the same channel at the same time. Devices that are hopping in a coordinated fashion can communicate with each other and are said to form a *piconet*. Every piconet has a single master and between 1 and 7 slaves. Communication in a piconet is strictly between a slave and a master. The channel-hopping sequence utilized by a piconet is pseudo-random, and can only be generated with the address and clock of the master device.

Every device implementing Bluetooth has a high resolution 24-bit clock (referred to as *CLKN* in the specification). This clock is used to keep the frequency hopping synchronized, as well as schedule other events. In order to participate in a piconet, the piconet master's BD_ADDR (a 48-bit MAC address) and clock must be known. Bluetooth device clocks increment at a rate of one every 312.5 microseconds.

Consider piconets in contrast to 802.11 networks. Two independent 802.11 APs would pick nonoverlapping channels to avoid collisions. Bluetooth devices avoid collisions not by picking a single channel, but by generating a pseudo-random list of channels and hopping rapidly through them. Because no two piconets will generate the same pseudo-random hopping sequence, on average, they will avoid collisions. For example, the hopping sequence illustrated in Figure 1 only contains one possible collision, at timeslot 3 on channel 5.

Device Discovery

Like all wireless protocols, Bluetooth has to handle the problem of determining whether potential peers are in range. This problem is significantly complicated by the frequency-hopping scheme just outlined.

Assume for a moment that a device is already interacting in a piconet (hopping along with its peers) and that it is also *discoverable,* which means that it wants to be found by other devices not already in its piconet. This means that it must be able to temporarily quit hopping

Device 1 and 2 form a piconet; they are channel hopping in step with each other.

Device 1 (master)	1	8	5	4	7	6	10	2	9	12	3	11
Device 2 (slave)	1	8	5	4	7	6	10	2	9	12	3	11

Device 3 is not part of the piconet; it is unaware of the channel-hopping sequence in use by the other devices.

Device 3	6	4	5	10	1	2	6	3	11	8	9	7

Figure 1 Three Bluetooth device-hopping sequences, two of which are communicating in a piconet

along with its piconet peers, listen for any devices that are potentially looking for it, respond to those requests, and then catch back up with the other devices in the piconet. Devices that periodically check for other devices looking for them are said to be "discoverable." Many devices aren't discoverable by default and must have this feature specifically enabled, usually for a brief period of time. Technically speaking, discoverable devices are devices that enter the inquiry scan substate. These devices respond to inquiry requests.

On the other end of this frequency-hopping dance is the device doing the discovery. This device has no knowledge of its potential peer channels at the moment, so it must transmit *discovery requests* (ID packets) into the air in a (mostly) random pattern, hoping to cross paths with a device on the same channel at the same time.

Even assuming that the discoverable device sees this request, how is it supposed to respond? It needs to transmit a response, but can't be sure what channel its discovering buddy wandered off to. Therefore, it will start responding on a lot of channels, on the assumption that its discovering buddy will see one of the responses.

This, in a nutshell, is the Bluetooth peer discovery process. It sort of reminds this author of the scenes in *Scooby Doo* when the gang is running through a hallway full of doors, being chased by a ghoul of some sort. All the Shaggy wants to do is find Scooby, but they never seem to cross paths. The same sort of humor can be appreciated in a Bluetooth discovery session.

Of course, this brief description is simplified; the protocol has a few optimizations to help devices find each other, and there is an upper-bound on the time it takes for this entire exchange to happen (10.24 seconds), but the process still seems remarkably difficult. If you've ever wondered what your computer was doing when it was looking for your cell phone or Bluetooth mouse the first time, this is it.

Also worth mentioning, a device is said to be nondiscoverable if it simply ignores (or doesn't look for) inquiry requests. The only way to establish a connection to one of these nondiscoverable devices is to determine its Bluetooth device address (BD_ADDR) through some other means. An example of this exchange appears in Figure 2.

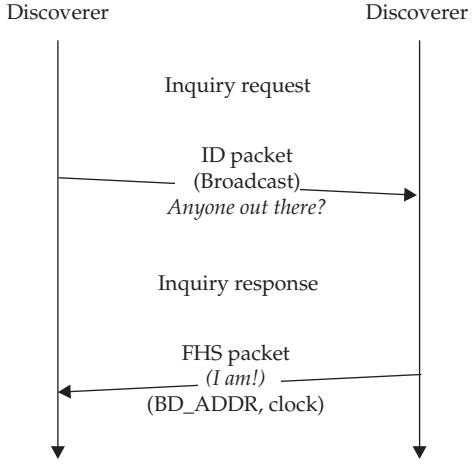

Figure 2 Bluetooth discovery process

Once the discoverer has the BD_ADDR and clock of the discoveree, it can then attempt to initiate a connection (send a page request), request the device's friendly name, and browse the advertised services over the service discovery protocol (SDP). We'll examine these protocols in more depth later in this chapter.

Connection Establishment

When a device wishes to establish a connection to another device, it must "page" it. This consists of transmitting a page request on the channel it thinks the target device is currently on. The transmitting device may not know the target channel for a number of valid reasons that include power savings, clock drift due to too much time passing since the last communications, and so on.

Devices that accept connection requests (*pages*) are said to be in *page-scan* mode, because they will periodically pause their current operation (such as relaying a real-time audio stream) to check to see if any other devices are interested in talking to them.

This entire process is very similar to the process used when performing device discovery, except that the frequency hopping is less of an issue since the device transmitting the page has a good idea as to the current channel the target device is on.

The diagram in Figure 3 covers this in some detail. The most important thing to remember about "paging" or connection establishment is that in order to establish a connection you must know the target's BD_ADDR, and that device must be interested in accepting connections.

Devices that aren't interested in accepting connections can only establish outbound connections. They avoid connection establishment by never entering the page-scan state. Devices that don't allow any inbound connections are called nonconnectable.

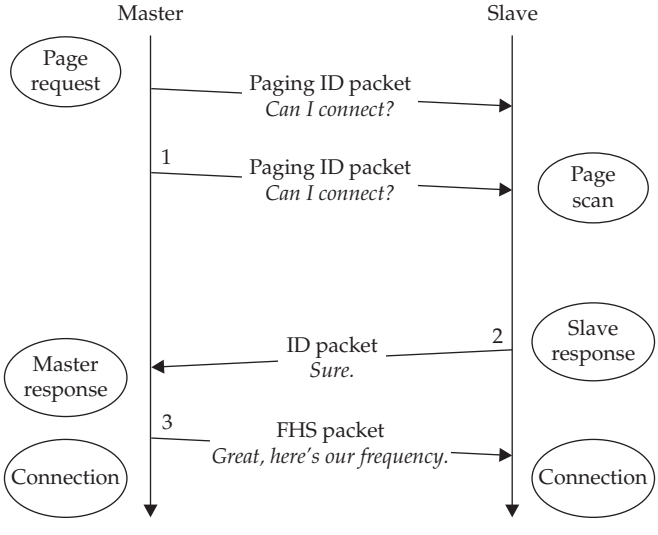

Figure 3 Connection establishment

Protocol Overview

A surprising number of protocols are used within a Bluetooth network. They can generally be broken up into two classes: those spoken by the Bluetooth controller and those spoken by the Bluetooth host. For the sake of our discussion, the *Bluetooth host* is the laptop that you are trying to run attacks from. The *Bluetooth controller* is sitting on the other end of your USB port, interpreting commands from the host.

Figure 4 shows the organization of layers in the Bluetooth stack and where each layer is typically implemented. The controller is responsible for frequency hopping, baseband encapsulation, and returning the appropriate results back to the host. The host is responsible for higher-layer protocols. Of particular interest is the Host Controller Interface (HCI) link, which is used as the interface between the Bluetooth host (your laptop) and the Bluetooth controller (the chipset in your Bluetooth dongle).

When dealing with Bluetooth, keep this host/controller model in your mind. As hackers, the thing we most desire over a device is control. The separation of power in the model shown in Figure 4 means that we are very much at the mercy of the Bluetooth controller. No matter how much we want to tell the Bluetooth controller "Stick to channel 6 and blast the following packet out forever," unless we can map this request into a series of HCI requests (or find some other way to do it), we can't. We just don't have that much control over the radio.

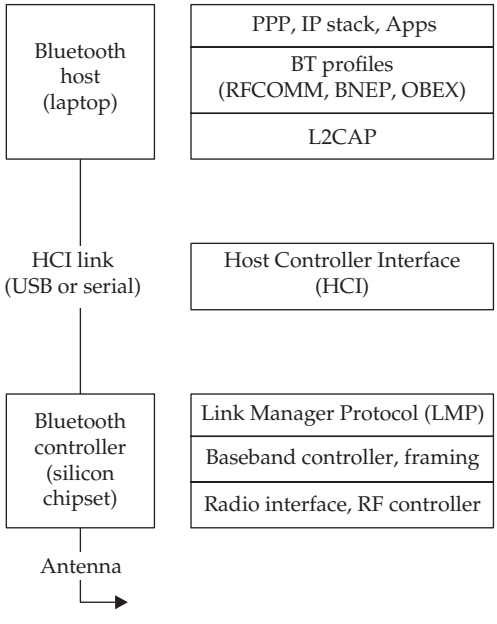

Figure 4 Bluetooth host/controller interaction

With the host/controller model in mind, let's cover the basic protocols utilized by the host when performing Bluetooth communication.

Radio Frequency Communications (RFCOMM)

RFCOMM is the transport protocol used by Bluetooth devices that need reliable streams-based transport, analogous to TCP. The RFCOMM protocol is commonly used to emulate serial ports, send Hayes AT commands to phones, and to transport files over the Object Exchange (OBEX) protocol.

Similar to TCP, RFCOMM has the notion of ports. Instead of 65,536 ports, however, RFCOMM has ports 1 to 30. In RFCOMM terminology, these ports are called *channels*.

RFCOMM is the most simple of the Bluetooth protocols to wrap your head around. It is also the highest level and most universally available to developers on restrictive platforms, such as mobile phones. RFCOMM is implemented on top of the L2CAP protocol, which we'll examine next.

Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP is a datagram-based protocol, which is used mostly as a transport to higher-layer protocols such as RFCOMM, SDP, and others. An adventurous application-level programmer can use L2CAP as a transport as well, and when used in this case, L2CAP has semantics similar to that of UDP (messaged based, not reliable, etc.).

L2CAP has a set of ports (independent from RFCOMM ports). Ports in the range 1–4,095 are reserved/well-known. Applications may use ports between 4,097 and 32,765. All L2CAP ports are odd. L2CAP terminology refers to these ports as *Protocol Service Multiplexers (PSMs)*.

Think of L2CAP as straddling the line between IP and UDP. Usually L2CAP is used to carry higher-level data packets; however, on some platforms, an application programmer can make use of it directly.

Host Controller Interface (HCI)

HCI is a protocol that has no allegory in an 802.11 or Ethernet-based network. As mentioned previously, the Bluetooth standard specifies an interface for controlling a Bluetooth chipset (controller). HCI is this interface. This technique means that much of the userland tools related to managing Bluetooth connections need no modification at all, even when a completely different Bluetooth chipset (controller) is used.

Controller Protocol Stack

The following protocols are handled by the Bluetooth controller (chipset). Unless utilizing specialized hardware, manipulation of these low-level protocols is outside the capability of mere mortals. As such, they are covered only briefly.

Asynchronous Connectionless Link (ACL)

ACL provides the lowest layer of encapsulation that a data packet will ever see before it reaches the baseband layer. ACL provides a transport layer for L2CAP packets. There is

virtually no programmer access to packets at this level as it is implemented on the Bluetooth controller.

Synchronous Connection Oriented (SCO)

SCO provides the encapsulation for low-fidelity (64 Kb/s) real-time streaming audio. This protocol basically exists to allow Bluetooth headsets to transmit audio at a predictable rate without worrying about competing for bandwidth with data packets. SCO connections have guaranteed terms of service while minimizing overhead. There can *only* be a single SCO connection between a master and a slave in the piconet.

While this protocol is listed in the controller stack, the host obviously needs to be aware enough to request using it for streaming voice-quality audio. This protocol does not provide enough bandwidth for streaming music in stereo.

The only time we will worry about SCO is when we are trying to decode audio from a Bluetooth headset.

Link Manager Protocol (LMP)

The Link Manager Protocol handles negotiation for such low-level issues as power-control, role-switching, QoS, and adaptive frequency hopping. It also handles encryption, authentication, and pairing. Although the controlling host may be aware of these features, and explicitly request them, the controller's job is to determine what sort of packets need to be sent, and how to handle the results. The LMP defines the format for these over-the-air packets. LMP packets are transmitted on the ACL link.

Baseband

The Bluetooth baseband specifies the over-the-air characteristics (such as the transmission rate) and the final layer of framing for a packet. Unlike 802.11, where receiving all the packets on a channel is trivial, actually getting a packet with in-tact baseband headers out of the controller and into the host is difficult. Even more difficult is handing the controller an arbitrary buffer and having it push this out to the air as a packet. Well-behaved programs simply have no need for such a feature, which means most silicon you can put your hands on (read: cheap Bluetooth dongles) simply doesn't implement it.

Figure 5 shows the organization of every Bluetooth packet at the lowest level.

Access Codes

The first field of every Bluetooth packet is the access code. When a Bluetooth controller receives a packet, the first thing it does is examine the access code to determine what to do. Access codes expand into the format shown in Figure 6.

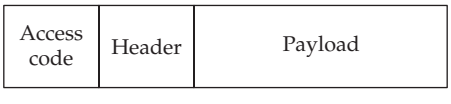

Figure 5 Basic Bluetooth packet

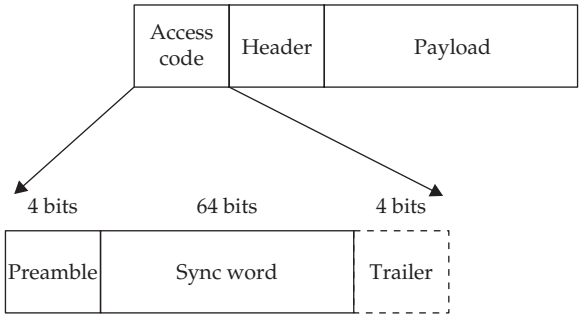

Figure 6 The bulk of the access code consists of the sync word.

As you can see, the bulk of an access code is taken up by a 64-bit sync word. This sync word is key to understanding how Bluetooth device addresses are used to establish a connection within a piconet.

The sync word is a 64-bit expansion of the lower 24-bits of the BD_ADDR that a device wishes to communicate with. This point is important enough to bear repeating: *The sync word of a Bluetooth packet is derived from the lower 24-bits of a device's BD_ADDR.* The details of this derivation can be found in section 6.3 of Volume 2 – Core System Package (Controller Volume) in the latest Bluetooth specification. Conceptually, you can think of the sync word expansion function as a hash, which has the very simple job of mapping 24 bits into a 64-bit space, although it is not designed to be cryptographically hard to reverse.

At any given point in time, a Bluetooth controller will be interested in only a handful of sync words. Any packets received by the controller with sync words that aren't interesting won't be passed through the HCI link. These packets are assumed to be for another piconet. At any given time, a particular Bluetooth controller will concern itself with three different types of sync words. These are covered next and in Table 1.

- The first access code that a Bluetooth controller will concern itself with is one with a sync word that corresponds with the local device's own BD_ADDR. Access codes of this type are called *Device Access Codes (DACs)* and are used to handle paging requests.
  When a Bluetooth controller sees a sync word that is an expansion of its own BD_ADDR, it means that someone is trying to page it. If the controller is connectable (meaning it accepts page requests), it will undergo the steps outlined in "Connection Establishment."
- The next case that a Bluetooth controller concerns itself with is when the sync word of an access code is derived from the BD_ADDR of the piconet's master. Access codes of this form are called *Channel Access Codes (CACs).* Packets with a CAC are used to carry application-level data, and the Bluetooth controller will need to examine the Logical Transport Address (LT_ADDR, the piconet-specific address) field of the header to determine if this packet is meant for the recipient, requiring further processing.

Sync Word Derived From	Used For	Name Given
Destination BD_ADDR	Channel signaling (paging requests)	Device Access Code (DAC)
Master's BD_ADDR	Data transport	Channel Access Code (CAC)
Reserved 0x9E8B00-0x9E8B3F	Inquiry (Device Discovery)	Inquiry Access Code (IAC)

Table 1 Access Code Derivation

- The final case that Bluetooth controllers are concerned with is that of *Inquiry Access Codes* (IACs). These codes are predefined in the specification and are used to indicate that a device is trying to discover other devices. There are sixty-four 24-bit predefined inputs to the sync word expansion function, which correspond to the sync words used for device discovery. As a consequence of this, there are 64 invalid values for the LAP of a BD_ADDR.

A Bluetooth controller that receives a packet that begins with an Inquiry Access Code will check to see if it is discoverable. If it is discoverable, the controller will respond to the request as outlined in "Device Discovery."

Bluetooth controllers utilize the sync word of every baseband packet to determine if it should be examined in more detail. Packets with sync words that the controller doesn't recognize are the result of another independent piconet operating nearby and are generally dropped.

Header Field

The Header field is expanded in Figure 7. Most of these fields aren't of concern to us unless we are implementing our own Bluetooth controller in software.

- **LT_ADDR** Logical Transport Address. Slaves in a piconet are numbered 1–7 dynamically when they join the piconet. This field is used to identify the source of transmission (when the packet is destined from the slave to the master), or the destination slave if the master is transmitting.
- **Type** The Type field is used to indentify the type of packet being used, indicating the data type (ACL or SCO), the data rate, and the number of slots it will occupy (where each transmission before the next frequency change is considered one slot).
- **Flow** This bit acts as a simple flow-control feature for packets transmitted over ACL (SCO packets have a guaranteed timeslot). When set to 1 (known as *GO*), the receiver has sufficient buffering space; 0 implies the opposite.
- **ARQN** Sometimes referred to as the sequence bit (SEQN), this field is used for positive acknowledgment of packet delivery and sequence numbering.
- **HEC** Header Error Check. An integrity check is performed over the entire packet; if the computed HEC at the receiver does not match the transmitted HEC, the packet is dropped. The HEC is a linear feedback shift register (LFSR) initialized with the UAP of the master.

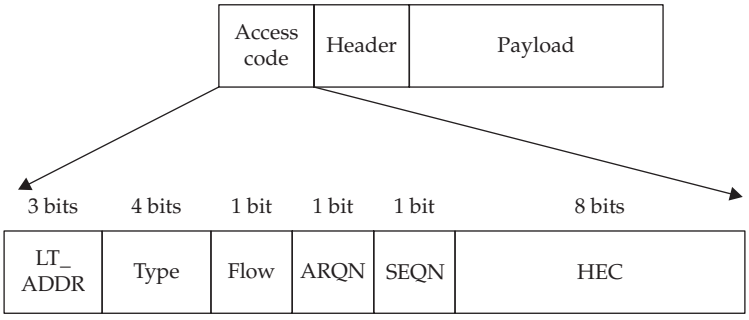

Figure 7 Expansion of the baseband header field

Bluetooth Device Addresses (BD_ADDR)

Bluetooth devices come with a 6-byte 802-compliant MAC address, similar to that of Ethernet and 802.11 devices. In Bluetooth, these devices have a little more structure to them and are rarely transmitted over the air. As outlined previously, the lower 24 bits of a BD_ADDR is expanded into a 64-bit sync word, which is, in turn, transmitted in the access code of a Bluetooth baseband packet. A BD_ADDR is composed of three distinct parts, shown in Figure 8.

- **NAP** The Nonsignificant Address Part consists of the first 16 bits of the OUI (organizationally unique identifier) portion of the BD_ADDR. This part is called nonsignificant because these 16 bits are not used for any frequency hopping or other Bluetooth derivation functions.
- **UAP** The Upper Address Part composes the last 8 bits of the OUI in the BD_ADDR.
- **LAP** The Lower Address Part is 24 bits and is used to uniquely identify a Bluetooth device.

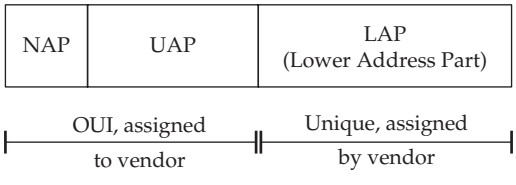

Figure 8 Details of a BD_ADDR

Bluetooth Profiles

If the previous whirlwind tour of the protocols involved in the Bluetooth stack weren't enough to get your head spinning, there are more than enough profile-level protocols to confound you. Roughly speaking, profile-level protocols would be equivalent to layer seven, application-level protocols. However, profile-level protocols can be layered on each other; therefore, not all profile protocols are on the same level.

Protocols at this level are implemented on the host and can be tinkered with freely. Profile-level protocols will not be covered in detail. Generally speaking, unless you are interested in fuzzing the applications themselves (not a bad idea, actually), you don't have to be concerned with the profile structure details. If you are, you can download the specifications and hopefully find some source code that implements it.

We should clarify that not *every* Bluetooth profile has its own profile-level protocol. Some profiles share an underlying protocol. Examples of this include the Intercom Profile (ICP) and the Cordless Telephony Profile, both of which make use of the Telephone Control Protocol (TCS-BIN).

With those caveats out of the way, here is very brief overview of the most popular Bluetooth profiles:

- **Generic Access Profile (GAP)** The most basic Bluetoooth profile, the GAP specifies things such as device names, PIN specifics, device classes, and security modes. Implementation of this profile is mandatory and ensures that various Bluetooth devices will always be able to communicate on some level.
- **Service Discovery Protocol (SDP)** SDP is utilized for one Bluetooth device to enumerate the services advertised by another Bluetooth device. Bluetooth services are identified with a name (required), and an optional list of attribute IDs, service classes, and profiles. These attribute IDs, service classes, and profiles allow services to identify themselves as printers, serial ports, etc. The Bluetooth specification allows quite a bit of detail to be encoded in these records.

  SDP's closest cousin in the traditional network world is probably the RPC portmapper. While the portmapper protocol doesn't provide such exhaustive descriptions of running services, it does map services to TCP and UDP ports. SDP is used in a similar manner—to map an advertised service to a particular RFCOMM or L2CAP port.
- **Advanced Audio Distribution Profile (A2DP)** Used for high-quality mono or stereo audio transmission. Used in Bluetooth headphones.
- **Headset Profile (HSP)** The profile that all of those Bluetooth headsets implement. Utilizes SCO for audio and a small set of Hayes AT commands for control.
- **Hands Free Profile (HFP)** This is basically an upgraded HSP, with a few more features such as last number redial. This profile is commonly found in high-end automobiles, although some more expensive headsets also support this profile.

- **Human Interface Device Profile (HID)** A wrapper for the popular USB HID protocol, which specifies an extensible protocol for mice, keyboards, etc. Wrapping the HID protocol allows code-reuse for developers.
- **Object Push Profile (OPP)** A simplified protocol that only allows the sender to transmit files. Implemented on top of OBEX.
- **Object Exchange Profile (OBEX)** This profile underlies many others, such as OPP, Basic Imaging Profile, and the Basic Printing Profile. The original OBEX profile was developed for infrared communication (IrDA), and specifies a generic way for two peers to exchange files.
- **Personal Area Networking Profile (PAN)** This profile allows for encapsulation of layer three network traffic over Bluetooth Network Encapsulation Protocol (BNEP).
- **LAN Access Profile (LAP)** A precursor to the PAN profile.
- **Serial Port Profile (SPP)** This profile Allows RS-232 emulation over RFCOMM.

This is just a small set of profiles that are supported by Bluetooth. For the curious reader, many of these profiles are available for download at *http://www.bluetooth.com/Bluetooth/Technology/Works/Profiles_Overview.htm.*

Encryption and Authentication

Encryption and authentication are built into the Bluetooth standard. Both are largely handled on the controller chip itself, not directly accessible to the Host Controller Interface layer.

Bluetooth devices can authenticate in two different ways: traditional pairing and the more recently added Secure Simple Pairing (SSP). SSP was added in version 2.1 of Bluetooth, but hasn't seen widespread adoption yet. Both pairing protocols are covered here.

For those of you anxious to move on to more concrete material, feel free to skim the details and dig into the next section where we dissect a basic Bluetooth adapter.

Traditional Pairing Process, Link Key Creation

This section covers the original traditional pairing process, which was superseded by the release of Bluetooth 2.1 and the Secure Simple Pairing process. The tradition process is still used in many Bluetooth devices.

In the traditional pairing process, when two devices connect for the first time, a link key is derived from a BD_ADDR, a PIN code, and a random number. The initial process is best described in Figure 9. Once both sides in the exchange have created K_init, they use it to generate a stronger link key. The derivation of the link key is described in Figure 10.

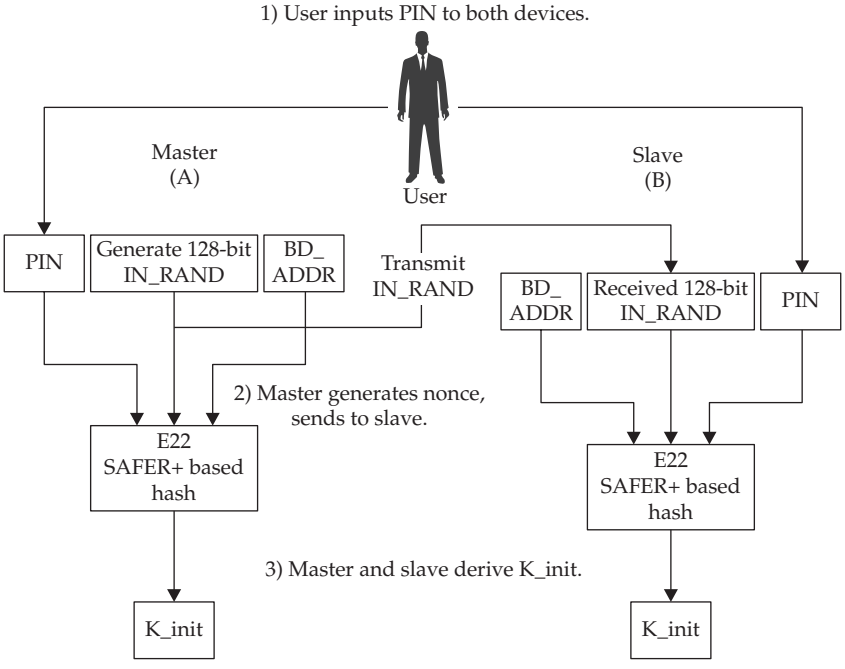

Figure 9 Initial link-key derivation step, resulting in K_init

At this point, both devices have generated a 128-bit link key. This link key will be stored alongside its peer's BD_ADDR, either on disk or in nonvolatile storage. When the two devices want to communicate, they will go through a handshaking process similar to that used in WPA. This handshake will prove possession of the link key. The link-key generation outlined here will only happen each time the devices are paired (typically once). For later authentication purposes, possession of this link key is used, not the PIN.

Proving Link-Key Possession

Bluetooth devices utilize possession of the link key to verify they are communicating with a device they have previously paired with. Proving possession of this link key authenticates the peer. The following authentication scheme is used to prove possession of the link key. The same exchange takes place if the link key was created using SSP or the traditional pairing technique and is illustrated in Figure 11.

In this exchange, there are two entities, the *Verifier* and the *Claimant*. The Verifier verifies that the Claimant has possession of the previously negotiated link key. The Verifier does this by transmitting a challenge in the clear, AU_RAND_a. Upon receiving

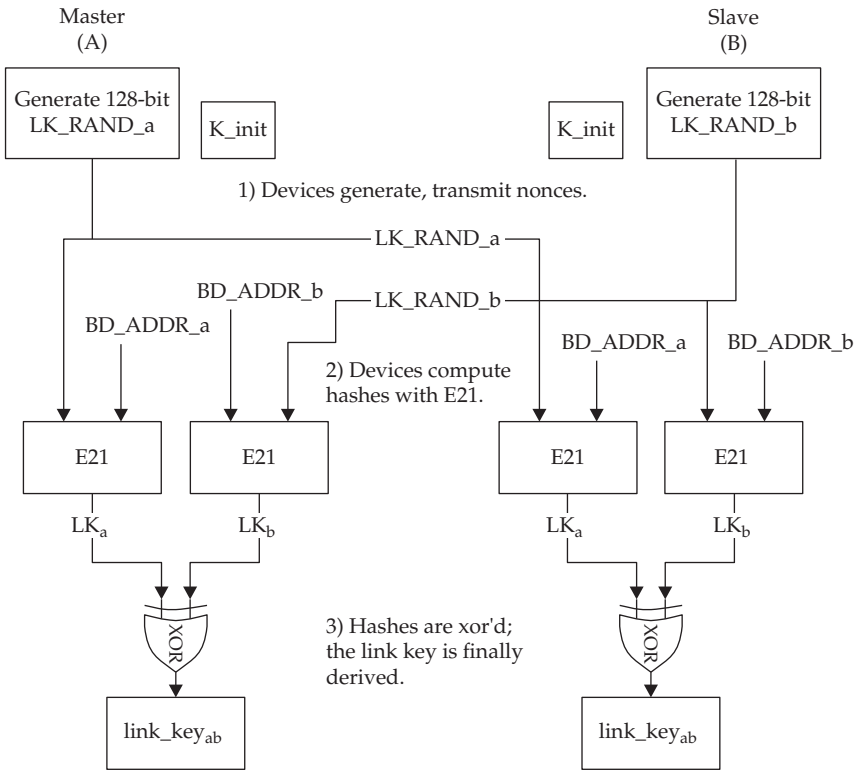

Figure 10 Link-key derivation

the challenge, the Claimant computes the keyed hash of the challenge using an algorithm called E1 and the link key. The Claimant transmits the results (called the *Signed Response,* or *SRES_c*) back to the Verifier. Meanwhile, the Verifier computes the same hash. If the Claimant's SRES_c matches the Verifier's computed hash, the Claimant has proven possession of the link key. If mutual authentication is desired, the two devices reverse roles and repeat the process.

Once the devices have authenticated each other by verifying possession of the link key, they will likely proceed to derive an encryption key. The encryption key is derived from a hashing function (called E3), which will take the link key as input. Details of keying the encryption algorithm can be found in the standard.

$link\_key_{ab}$ was derived during the pairing process.
This portion of the protocol is very similiar to the four-way handshake used by WPA; instead of verifying possession of the PMK, we are verifying possession of the link_key.

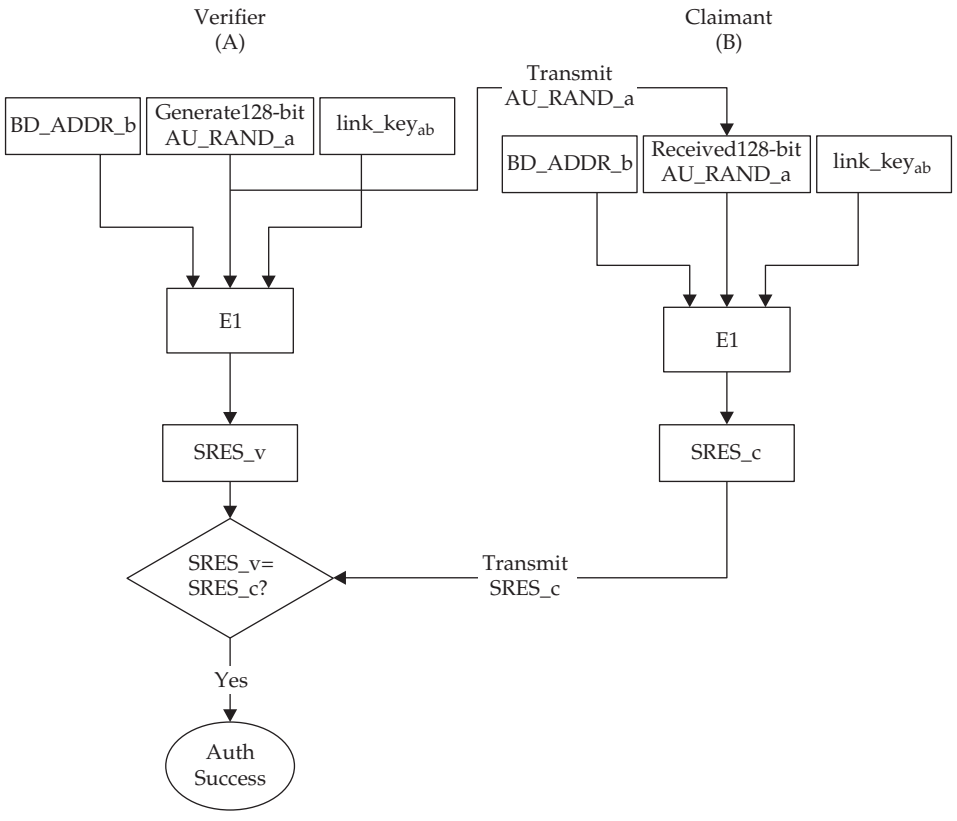

Figure 11 Authentication of link-key possession

Note

If an attacker observes the traditional link-key generation step, as well as the link-key authentication step, all he needs to do to derive the link key is to brute-force the PIN until he generates the observed value of SRES. Once he gets the SRES to match, he possesses both the PIN and the link key. Details on how to mount this attack are covered in Chapter 10 of the book.

Secure Simple Pairing

The biggest problem with the traditional pairing scheme just outlined is that a passive attacker who observes the pairing exchange can typically brute-force the PIN in seconds. Secure Simple Pairing attempts to prevent a passive observer from retrieving the link key.

SSP accomplishes this by using public key crypto, specifically *Elliptic Curve Diffie-Hellman*. A Diffie-Hellman key exchange allows two peers to exchange public keys and then derive a shared secret that an observer will not be able to reproduce. The resulting secret key is called the *DHKey*. Ultimately, the link key will be derived from the DHKey.

By using a Diffie-Hellman key exchange, a strong pool of entropy is used for deriving the link key, solving the biggest problem with the standard pairing derivation, where the sole source of entropy was a PIN only a few digits in length.

SSP adds another layer of authentication to the pairing process. In the case of SSP, you are trying to authenticate that the device you engaged in a Diffie-Hellman key exchange is the device you think it is. Doing this depends on the devices both having some form of input and output you can use to communicate with. These IO capabilities are explicitly negotiated during the SSP exchange.

The entire pairing exchange can be broken into stages. These stages are covered next and shown in Figure 12.

Capabilities Exchange

The first stage in an SSP session is a capabilities exchange. During this exchange, the link managers on both devices trade information on whether they have the capability to input

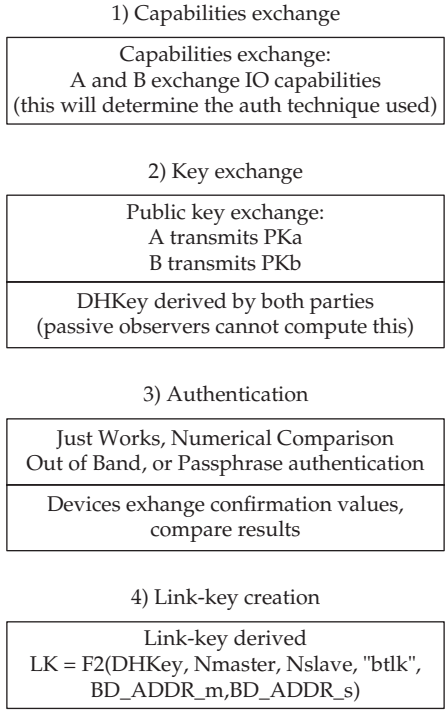

Figure 12 Secure Simple Pairing overview

Capability	Description
No input	Device cannot indicate yes or no, lacking any input capability.
Yes/No	Device has a button that the user can activate to indicate yes or no.
Keyboard	Device can input values 0–9, as well as indicate yes or no.

Table 2 Input Capabilities

Capability	Description
No output	Device cannot display a 6-digit number.
Numeric output	Device can display a 6-digit number.

Table 3 Output Capabilities

Local Input/Local Output	No Output	Numeric Output
No input	NoInputNoOutput	DisplayOnly
Yes/No	NoInputNoOutput	DisplayYesNo
Keyboard	KeyboardOnly	DisplayYesNo

Table 4 Merged Input/Output Capabilities

or display information. This exchange is important because the Just Works authentication method, which is used when there is no better option, doesn't provide protection against active man-in-the-middle (MITM) attacks. A table summarizing the capability information exchanged by devices is included in Tables 2 and 3.

These separate input and output capabilities are mapped into a single IOCapability value, as defined in Table 4.

Some combinations of IOCapabilities cannot provide a defense against a MITM attack. When a link key is generated and the authentication used *can* protect against MITM attacks, the device key is called "authenticated." If the authentication scheme cannot protect against MITM attacks, the link key is said to be "unauthenticated." The possible combinations of IOCapabilities, as well as the resulting authentication status, are produced in Table 5.

Surprisingly, many devices may be happy to use the weaker form of "Just Works" authentication, *even when both devices support a more secure authentication scheme*. This is because devices use the following algorithm to determine which authentication system to use.

1. Has any Out-Of-Band (OOB) data successfully been received from the other device? Then use the OOB authentication technique.

B Resp/ A Initiator	Display, YesNo (display + btns)	DisplayOnly (display, no btns)	Keyboard only (No output)	NoInputNoOutput (No anything)
Display, YesNo	NumberCmp: Authenticated	NumberCmp: Auto-conf, Unauthenticated	Passkey entry: Authenticated	NumberCmp: Auto-conf, Unauthentic
Display Only	NumberCmp: Auto-conf, Unauthentic	NumberCmp: Auto-conf, Unauthentic	Paskey entry: Authenticated	NumberCmp: Auto-conf, Unauthentic
Keyboard Only	Passkey entry: Authenticated	Passkey entry: Authenticated	Paskey entry: Authenticated	NumberCmp: Auto-conf, Unauthentic
NoInput NoOut	NumberCmp: Auto-conf, Unauthentic	NumberCmp: Auto-conf, Unauthentic	NumberCmp: Auto-conf, Unauthenticated	NumberCmp: Auto-conf, Unauthenticated

Table 5 IOCapabilities and Authentication Status

2. Do *both* devices support insecure (unauthenticated) link keys? Then use the Just Works Numeric Comparison with no user confirmation.
3. Does either device require an authenticated link key? Then choose the appropriate scheme from Table 5.

Notice that if both devices are configured to accept unauthenticated link keys, they will do so, even if they both have extensive IOCapabilities.

Once the IOCapabilities exchange has taken place, the devices should know what authentication technique will be used later. The next step is to exchange public keys and derive the DHKey.

Key Exchange

Key exchange is the easiest phase. The devices simply transmit their public keys to each other and then perform a Diffie-Hellman key exchange operation to derive DHKey. An attacker who captures this entire exchange will still be unable to compute DHKey, due to the cryptographic security of the Diffie-Hellman protocol. This key exchange is shown in Figure 13.

Authentication

Once the DHKey has been derived, there are four possible authentication techniques. Keep in mind that, although these are called "authentication" techniques, they are authenticating something entirely different than the authentication scheme you saw in traditional pairing. In traditional pairing, you are verifying *possession* of a link key. At this current point in the SSP exchange, you haven't created the link key yet. Instead, you are trying to verify that the

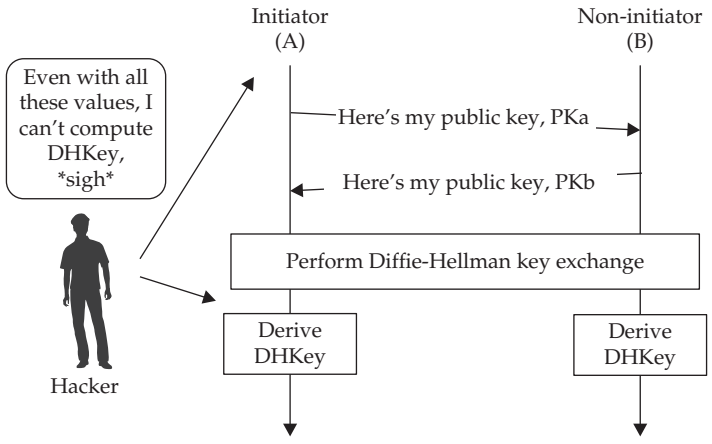

Figure 13 Initial key exchange in an SSP

device you just performed a Diffie-Hellman key exchange with is the device you think it is. Ultimately, verification of the link key will take place using the same algorithm used in traditional pairing.

- **Just Works** This mode runs the same protocol as numeric comparison, but the user does not actually make a comparison. This protocol is secure against passive attacks (due to the DHKey exchange), but an active MITM attack can succeed by sending both A and B its own public key and nonces at the right time.
- **Numeric Comparison** If both devices have sufficient IO capabilities to display a six-digit number to the user, Numeric Comparison is used. When this technique is used, each device computes a hash of the exchanged public keys, as well as two more nonces. The devices display six digits of this hash, and the end user is expected to verify that the displayed hashes match and select Yes or No. Although this may *appear* to be similar to the traditional pairing process, cryptographically it is very different. In this case, the six-digit values displayed to the user are an artifact of the pairing process. They are not used as input to any cryptographic functions. Also, in this mode, the user is *not inputting* the number, rather he is *comparing* the number, which is a hash generated by both devices.
- **Out of Band** This mode is used if the devices have some way other than Bluetooth to exchange cryptographic material. Near Field Communication is described as one possibility. If the OOB communications technique can be exploited with a MITM attack, this authentication technique will be similarly vulnerable.
- **Passkey Entry** Passkey entry allows Bluetooth devices to authenticate each other by verifying they both possess a shared secret. This secret can be entered into both devices or generated on one and entered into the other. A typical use-case for this

is a keyboard and computer. The computer generates a random PIN, and the user inputs it through the keyboard. When implemented poorly, this technique can be severely compromised.

The Just Works, Numeric Comparison, and Passkey Entry techniques are believed to receive the greatest adoption in next-generation Bluetooth devices and are covered in more detail.

Just Works Technique The Just Works authentication technique was a necessary accommodation to achieve the greatest level of compatibility between electronics design and Bluetooth security. While other authentication mechanisms offer a greater level of security, they all require some level of Man-Machine Interface (MMI) to display content or collect responses from the end-user. This requirement would otherwise limit the scope of deployment options for Bluetooth developers, requiring an authentication mechanism that does not require any input from the user.

As a dissatisfactory by-product, the Just Works mechanism is also the easiest to use, leading developers and end-users to choose this authentication mechanism over stronger protocols. Although the Just Works method protects against the passive eavesdropping attacks that plague legacy PIN authentication, it does not protect against active attacks. Devices that accept Just Works authentication are incapable of authenticating the identity of the remote entity or defeating MITM attacks, leaving the link-layer exposed to a variety of upper-level profile attack options.

For example, consider a case where a Bluetooth USB HID interface is used on a PC intended for use with a Bluetooth keyboard. If the PC is connectable, an adversary could impersonate a legitimate keyboard device and send arbitrary keystrokes to the host, perhaps downloading and running malware-ridden executables from the Internet. In this example, the PC device is at fault for accepting the Just Works authentication technique when stronger input mechanisms are available, though it isn't unreasonable to foresee a situation where Just Works is used by a device manufacturer to simplify the product setup process.

Numeric Comparison Technique In the Numeric Comparison technique, both devices generate and exchange nonces (a number used once), and then compute a hash of these nonces as well as the public keys exchanged previously. This hash is displayed to the user in the form of a six-digit number. The user is supposed to compare these numbers to verify that they match. If so, the user presses the Yes button to indicate pairing should proceed. If the hashes don't match, then an active attacker has tried to inject keying material or a transmission error has occurred. In either case, the pairing shouldn't proceed. This process is shown in Figure 14.

Passkey Authentication Technique In the Passkey authentication mode, the user inputs an identical passkey into both devices, or one device generates a passphrase which is input to the other. The goal of the protocol is to verify the passphrase with the other device, one bit at a time. For every bit in the passphrase, the devices will compute a hash including the bit and transmit the hash to the other side. In the simplified example shown in Figure 15, both sides will transmit eight hashes.

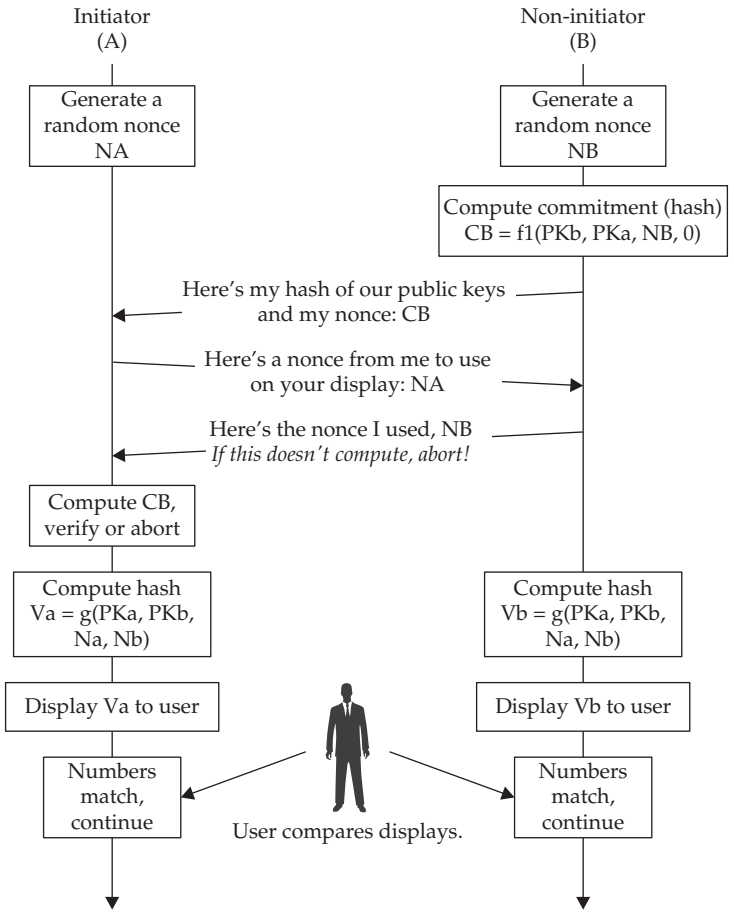

Figure 14 Numerical Comparison authentication technique

The Bluetooth specification calls this protocol a "gradual release" procedure of the passphrase, and it is rationalized in the following manner. Device A reveals ra[i] before device B does. B, however, has already transmitted his commitment hash, before A reveals the bit. This means that B cannot change his choice at this point (in other words, he is *committed*).

This process prevents a MITM attack because the commitment hash also has the public keys used in the DHKey exchange as input. A potential MITM is, therefore, in the difficult position of having to transmit a hash, which has an input the attacker will only be able to guess with 50 percent accuracy. The odds that an attacker will successfully guess a randomly generated passphrase in such a matter are $½^i$, where i is the length of the passphrase in bits.

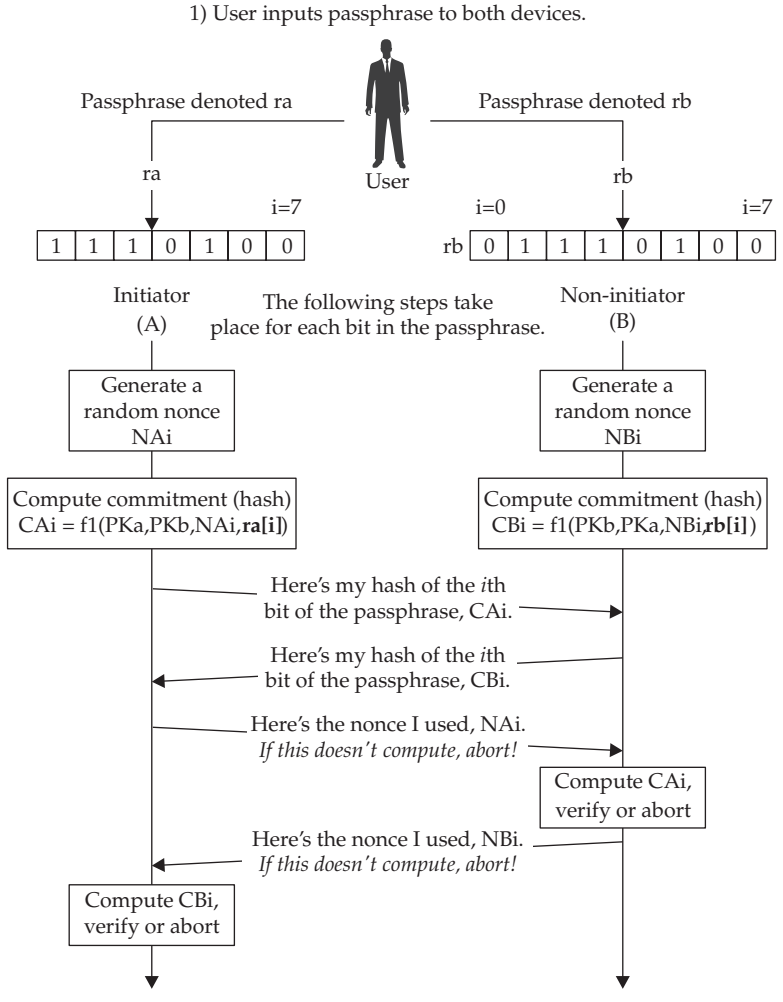

Figure 15 Passkey authentication

Passively Attacking Passphrase Authentication

Popularity:	*2*
Simplicity:	*4*
Impact:	*6*
Risk Rating:	***4***

An attacker who can observe the entire exchange shown in Figure 15 can trivially recover the passphrase. For every bit in the passphrase, the attacker will know the following values:

PKa, PKb	Public keys observed during the initial public key exchange
CAi, CBi	The commitment hashes for the *i*th bit
NAi, NBi	The nonces used as input to the above commitment hashes

Given these values, an attacker has everything he needs to compute the *i*th bit himself. Remember, there are *only two possible values* for ra[i]: 1 or 0.

The attacker simply has to compute f1(PKa, PKb, Nai, **0**). If the result equals CAi, then ra[i] = 0. If that doesn't pan out, the attacker can compute f1(PKa, PKb, Nai, **1**), which will reveal ra[i] is, in fact, 1.

The gist of this is that an attacker can retrieve the passphrase in its entirety, and all he has to do is observe the pairing and compute a few SHA256 hashes.

In this regard, SSP is actually worse than the PIN-based scheme used in traditional pairing. Under the old system, if the user inputs a 32-bit pin, an attacker will need to compute 2^{32} hashes (worst case) before finding it. In the current system, the attacker will need to compute at worst 32 hashes.

The saving grace of this technique is that an attacker who learns the passphrase still does not know the link key, because it will be derived from the DHKey, which an attacker cannot derive. Once the attacker has recovered the passphrase, he can try to convince the devices to pair with him.

Actively Attacking Passphrase-Protected Devices

Popularity:	*2*
Simplicity:	*2*
Impact:	*6*
Risk Rating:	***3***

Another attack can be levied against a passphrase-protected device. Assume that a device has a 32-bit passphrase (for simplicity). Also assume that this device can be placed in pairing mode repeatedly. Finally, assume that you would like access to this device, and you don't have the link key or passphrase.

All you need to do is randomly choose a bit for ra[i], starting with ra[0] and working your way up. If the device continues, then you chose correctly. If the device aborts, you know that you chose incorrectly and, therefore, will choose correctly the next time.

Assuming the device has a 32-bit passphrase, you will be able to guess the entire passphrase correctly after 16 pairing attempts (on average). The problem is that the protocol reveals a bit of the passphrase to the attacker at every step, regardless of her choice. The specification says that exponential back-off times should be used to slow down attacks such as this, but when such a small number of attempts are needed, it seems unlikely to help.

Countermeasure

The moral of this story is that devices using passphrase authentication should *never* be configured to reuse a static passphrase. If the passphrase is randomly generated on every attempt, you will not be able to carry the partial information across attempts.

Authentication Phase 2

Regardless of the authentication technique used, once authentication has completed, the following exchange takes place. Device A computes a hash over all of the important values used previously (the DHKey, IOCapabilities, BD_ADDRs, etc.) and transmits the hash to device B. This is device B's last chance to verify that they have agreed on everything so far. Device B should verify this hash, and if it checks out, send a similar hash of his own. At this point, the authentication phase is complete. This process is shown in Figure 16.

Link-Key Derivation

Once the authentication phase is complete, the devices are as convinced as they ever will be that the DHKey was negotiated with the desired party. Now it's time to use it for creating

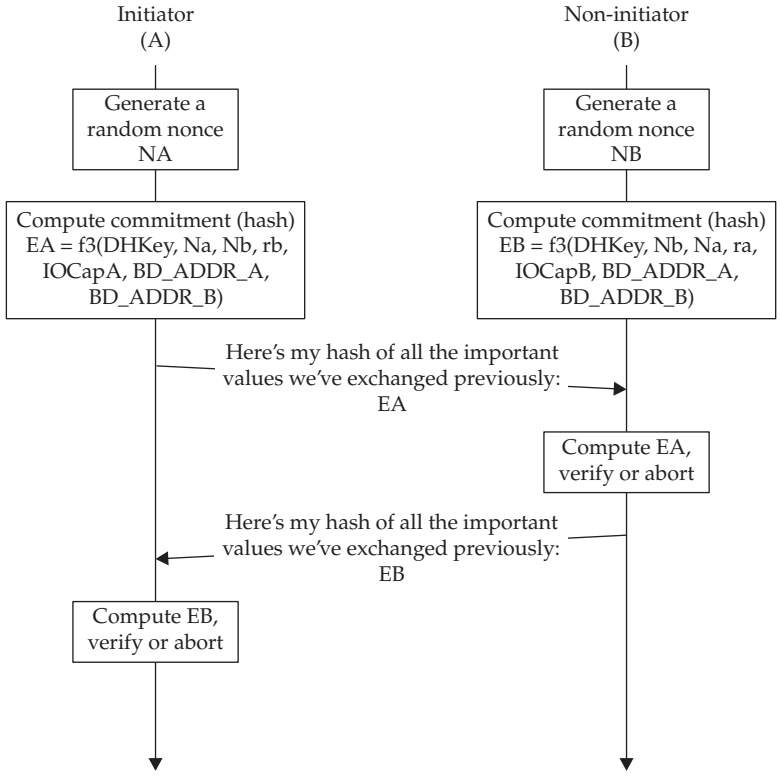

Figure 16 The last stage of authentication, which is common among all authentication techniques

the link key. Link-key derivation is simple at this point because all of the authentication is out of the way.

The link key is derived from the DHKey using the following hash.

```
LinkKeyAB=f2(DHKey, Nmaster, Nslave, "btlk", BDADDR_master, BD_ADDR_slave)
```

Once the devices compute the link key, they will store it along with the peer's BD_ADDR. Then they can use it for future authentication sessions, without having to re-create it.

At this point, Secure Simple Pairing becomes identical to traditional pairing. Possession of the link key is performed utilizing the same technique shown earlier in Figure 11. Encryption keys will be derived using the same hashing values. The same encryption system is used to protect the link. Note that an attacker who observes any of these exchanges, and then observes the exchange in Figure 11 will *not* be able to compute the link key because it was derived from a Diffie-Hellman key exchange.

SSP Summary

Despite attacks on passphrase authentication just outlined, on average Bluetooth security is enhanced via the use of SSP. Unless there is a serious cryptographic breakthrough, a passive attacker should not be able to recover the link key since this would require a passive attack against the Diffie-Hellman key exchange.

The best passive attack known to date involves the misuse of the passphrase authentication scheme with a static key. An attacker who observes this can trivially compute the passphrase used for the pairing session. This is a serious flaw, because the attacker can potentially pair with the compromised devices herself, but she still cannot impersonate one device to the other, or decrypt intercepted traffic.

In order for an attacker to recover the link key between two devices, she must actively attack them during the pairing process. A successful attack will result in the compromised devices pairing with the attacker, instead of themselves. The attacker can then read/write data to both devices. The simplest such MITM attack involves falsifying IOCapabilties, and is known as the *Niño* attack.

SSP Niño Attack

Popularity:	*2*
Simplicity:	*4*
Impact:	*6*
Risk Rating:	***4***

Recall from our evaluation of the capabilities exchange that, when two devices pair, they negotiate the IOCapability information to identify a suitable authentication mechanism that is supported by both devices. This exchange happens before the link key is derived and, as such, is not a protected exchange.

Documented in his doctoral thesis from the University of Kuopio, Finland, Keijo Haataja describes the Niño attack, also known as the NoInputNoOutput attack. This attack leverages

this SSP IOCapabilities exchange deficiency to manipulate one or more devices, forcing the victim device to "dumb-down" its selected authentication mechanism to the Just Works technique. Once one or more devices are forced to this weaker authentication method, the attacker can eavesdrop on any data sent between devices.

First, the attacker must force two devices to re-pair. One option is to launch a denial-of-service (DoS) attack against the Bluetooth devices in the area, designing a transmitter that hops along with the piconet master and jamming on each channel during the frequency-hopping exchange. A second option is to leverage a wide-band jammer that can jam all 79 Bluetooth channels simultaneously, ceasing all Bluetooth communication within range of the attacker. Once the end-user becomes frustrated with the lack of communication between two Bluetooth devices, the attacker would hope that the user attributes the failure to the devices themselves and attempts to repair the devices to resolve the issue, at which time the attacker would stop the DoS attack.

Immediately before the devices re-pair, the attacker would impersonate the BD_ADDR and friendly name of both devices and implement a MITM attack, brokering the authentication exchange between the victim devices. Instead of allowing the legitimate advertised IOCapabilities to pass between devices, however, the attacker would indicate to the responder that the initiator only supports the NoInputNoOutput capability, and vice versa, effectively dumbing-down the connection exchange and leaving the Just Works authentication method as the only plausible method.

Because the Just Works method does not provide protection against MITM attacks, the attacker can complete the authentication process with each device to conclude the pairing exchange and finish the connection establishment. With knowledge of the link key derived by both devices, the attacker can now eavesdrop on the traffic between the devices, optionally manipulating the content as desired.

Countermeasure

The Niño attack leverages a design concession in the SSP specification to accommodate devices with no man-machine interface. Due to the lack of cryptographic integrity protecting the IOCapabilities exchange, an attacker could leverage this weakness to manipulate one or more victim devices into creating a connection with the attacker.

At the time of this writing, there are no available exploit tools to implement the Niño attack. With integrated support in the BlueZ stack for SSP, however, we believe it would be possible to impersonate both devices in anticipation of a pairing exchange and force the use of the Just Works authentication mechanism. Additional research and experimentation is needed to evaluate the practicality of this attack against real-world implementations.

One significant requirement for the Niño attack is to force a Bluetooth user to delete the prior pairing information and re-pair devices, or to actively catch a user pairing two devices for the first time. You can leverage this attack requirement as a countermeasure, by not pairing Bluetooth devices in any location that is susceptible to traffic sniffing attacks. If one or more of your Bluetooth devices prompts you to spuriously repair, disable Bluetooth on both devices temporarily until you can return to a safe location and re-pair.

Finally, as a reward for wading through all of the abstract protocol descriptions, here's something concrete. The next section covers the dissection of a Bluetooth dongle.

Anatomy of a Bluetooth Adapter

Bluetooth adapters share many of the same important characteristics as 802.11 cards. Antenna connectors, receive sensitivity, and transmit power are obviously important. The most important thing about a Bluetooth adapter, however, is its chipset. To better understand why this is, we are going to dissect a commonly available Bluetooth adapter, the D-Link DBT-120 HW ver C1. The reason we chose this adapter is because it's easy to find and is known to reliably contain a Cambridge Silicon Radio (CSR) BlueCore4-external chipset. This chipset is desirable because it is well documented and well supported under Linux, and it has an external flash chip for firmware storage, which means it is easily hackable.

If you take apart a DWL-120, you will find a board that looks like the one shown in Figures 17 and 18.

Though too small to make out, the chip in Figure 17 is labeled "BC417 143BON 751AA." This is the CSR BlueCore4 chipset. Figure 18 shows the other side of this board, which contains a small flash chip. This flash chip is where the firmware for the BC4 chipset is stored. The particular model found in this author's adapter was in the M29W800 family from STMicroelectronics.

By looking at the two images, you can determine that the entire D-Link adapter consists of a BC4 chipset, some flash memory, a small PCB with a crystal, and a USB plug. Essentially, all that D-Link did was buy a BC4 chipset from CSR, put it on a board with some flash, wrap it in plastic, paint it orange, and sell it to us. Seeing as how this adapter is entirely defined by its chipset, let's dig a little bit deeper into the BC4 external chip.

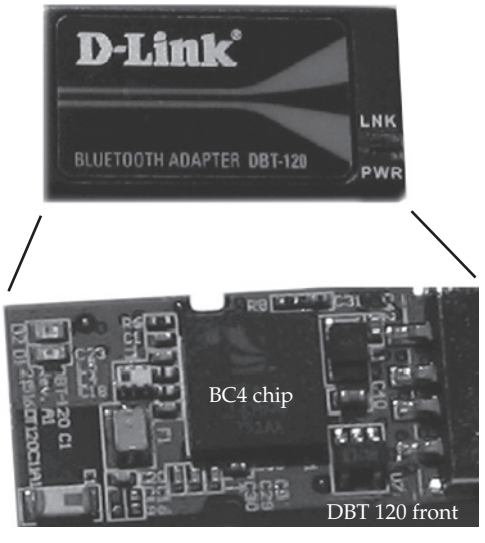

Figure 17 The front of a CSR BC4-dongle. The BC4 chipset is the large chip in the middle.

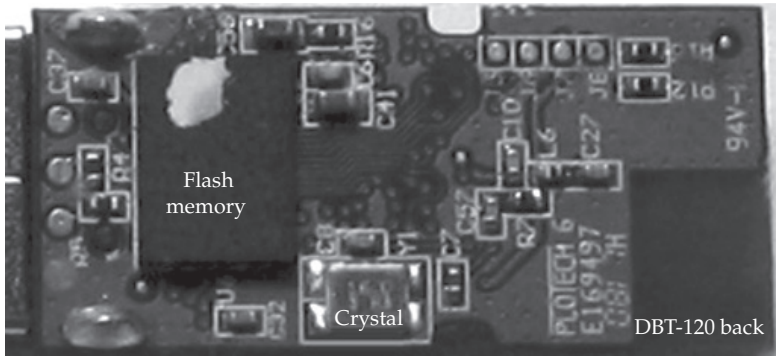

Figure 18 The back of a CSR BC4-dongle. The flash storage is the only chip on this side.

BlueCore4 Block Diagram

Figure 19 is a simplified version of the diagram found in the BC4 datasheet. Only the aspects most relevant to us as hackers are included.

As you can see, the BC4 chip contains a memory manager, USB interface (which our HCI commands flow through), a UART (also a potential HCI interface), a Synchronous

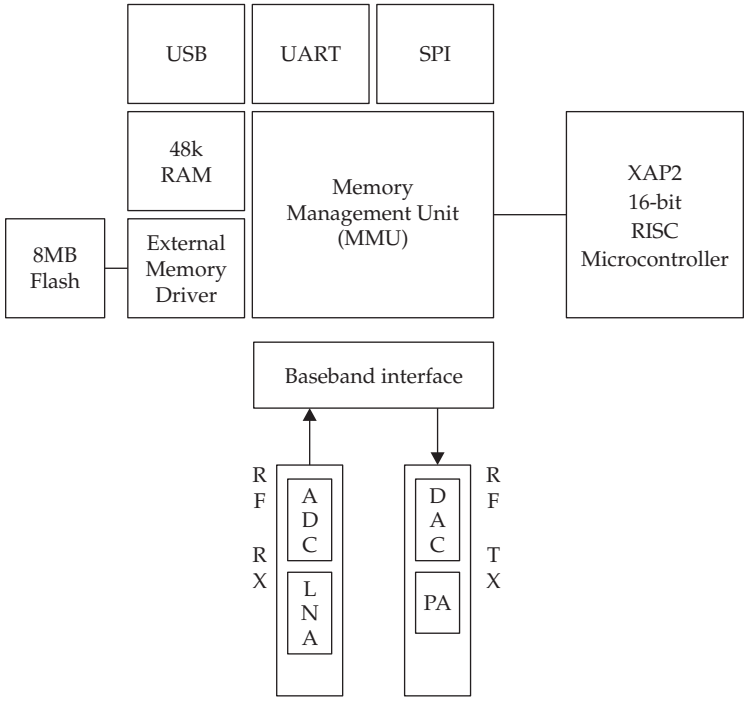

Figure 19 Simplified block diagramof BC4—external

Serial Protocol Interface (used for developers), a flash interface, a 16-bit microcontroller, and, of course, all of the logic necessary for receiving and transmitting Bluetooth packets. You basically have a tiny general-purpose computer, complete with 48K of RAM, 8MB of storage, and a 16-bit CPU. Not bad for about $25.

One of the most interesting things to know about the BC4 chip is that it actually provides support for running third-party code *on the chip itself*. The details of this are outside the scope of this Bonus Web Chapter, but, in short, CSR provides a software development kit (SDK) called Casira that allows other vendors to compile and upload code to a BC4 device. The uploaded applications run on a virtual machine. This SDK is available for $3,000 directly from the CSR store (product id: *BLUELAB*).

Unfortunately, this $3,000 price tag is enough to keep open source hackers at bay, as we are currently unaware of any open source firmware developed for the BC4 architecture. However, the extendibility of the BC4 chipset will feature prominently in our efforts to gain control over a Bluetooth controller in Chapter 9.

Summary

This concludes our tour of the Bluetooth protocol stack. Figure 20 provides a summary of the Bluetooth protocol stack for reference. With this legwork out of the way, you are well prepared to dig into the exciting world of Bluetooth device hacking discussed in Part II of the book.

Definitions

Access code The first 68 to 72 bits of every baseband packet transmitted by Bluetooth. A sync word takes up the bulk of an access code. These come in three flavors: CACs, DACs, and IACs.

Basic data rate The original 1 Mbps data rate.

BD_ADDR Bluetooth Device Address. Similar to a MAC address in that it is 48 bits, but differs significantly in that only 24 bits are used in the baseband packet header.

Channel Access Code (CAC) Handles data delivery. The sync word in this code is derived from the lower 24 bits of a piconet master's BD_ADDR.

Commitment A SHA-256-based hashing function used in Secure Simple Pairing sessions.

Device Access Code (DAC) Handles paging. The sync word in this code is derived from the lower 24 bits of the destination device's BD_ADDR. Without a target device's BD_ADDR, its corresponding DAC cannot be derived. Without its DAC, a page request cannot be sent and, therefore, no data transmission can take place.

Discoverable A device that wants to be found is called "discoverable." Technically speaking, a device is discoverable if it responds to inquiry scans.

Enhanced Data Rate (EDR) The improved 2 and 3 Mbps data rates.

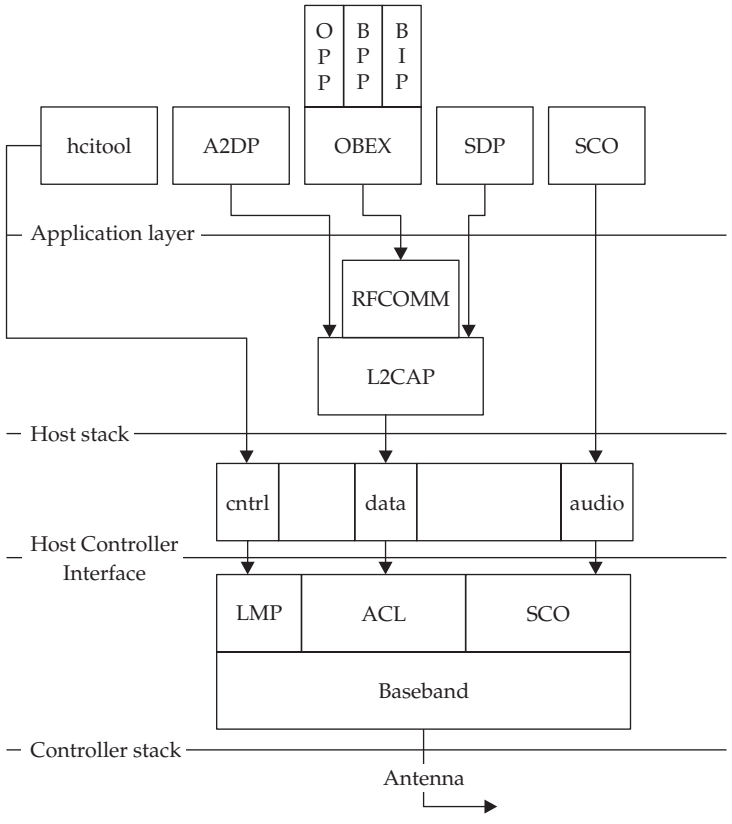

Figure 20 An overview of the Bluetooth protocol stack

FHS A Frequency Hopping Synchronization packet is a special packet in the Bluetooth world, containing the full BD_ADDR and clock of the transmitting device during the paging exchange.

Inquiry Access Code (IAC) Handles inquiry signaling. The sync word in this code is used to indicate whether a device is searching for any Bluetooth device in range or for a particular class of devices

LAP Lower Address Part. The lowest 24 bits of a BD_ADDR.

Master Controller of a Bluetooth piconet. The master's clock and BD_ADDR determine the piconet-hopping sequence.

NAP Nonsignificant Address Part. The 16 bits following the UAP.

Page What most people would call a "connection," Bluetooth calls a page. A device that is attempting to establish a connection with another device is said to "page" it.

Page scan Devices that are willing to accept inbound connections respond to page requests by periodically entering the Page scan state.

Piconet A set of Bluetooth devices hopping in sequence with each other. These devices can communicate among themselves since they are always on the same channel at the same time.

Slave Participant in a piconet. The slave's clock does not drive the hopping sequence.

Sync word Used by Bluetooth controllers to determine if a packet should be processed or dropped. Sync words are 64-bit expansions of a device's LAP and are transmitted in access codes.

UAP Upper Address Part. The high 8 bits of a BD_ADDR, usually 00.