CVE-2020-12586: A Silent Pairing Issue in Bluetooth-based Contact Tracing Apps

Authors: Alwen Tiu (The Australian National University) and Jim Mussared (George Robotics)
May 2020
Last modified: 18/05/2020

Summary

We have found a way in which several contact tracing apps allow an attacker to:

- obtain a permanent identifier of the Android phone running the app,
- perform a silent pairing between the attacker controlled device with the target phone, and
- permanently detect the presence of that device even after the app is uninstalled.

The barrier to create a working exploit is low; only a moderate level of knowledge of bluetooth programming is required. The impact on privacy is high.

To launch the attack the attacker needs to be within the Bluetooth range of the target phone but does not require the attacker to have physical access to the phone. It works silently, without prompting the user of the target phone to initiate pairing. Once pairing is complete, the permanent and unique Bluetooth MAC address of the phone (also called the ‘public address’ or ‘identity address’ in Bluetooth lexicon) is revealed, and a cryptographic key, called the Identity Resolving Key (IRK), is sent to the attacker. The IRK is used in Bluetooth Low Energy (BLE) devices to produce random addresses (also called ‘resolvable private addresses’) that are meant to prevent tracking of the smartphone. An attacker in the possession of the IRK will be able to defeat this protection mechanism to uniquely identify the BLE device used in the user's phone.

This affects all versions of Android that we tested, including the latest release (versions 6.0 to 10.0). We also have shown that once the identity address of the phone is known, then, depending on Android version, other attacks become possible, leading to permanent tracking of the device (even when the contact tracing app is uninstalled), denial of service, remote control of the phone, and potentially remote code execution.

Additionally, a non-silent variant of this is possible on iOS devices (requiring the owner of the phone to agree to a pairing prompt), however the pairing request is extremely innocuous as the text is mostly attacker-controlled. If successful, the same permanent tracking of the device becomes possible.

Overview of contact tracing protocol implementations

Several contact tracing apps use a Bluetooth Low Energy (BLE) based implementation to detect proximity and exchange temporary IDs (TempIDs) to log whom they have been in close contact with. Irrespective of whether an app implements a centralised or a decentralised approach, it would need to employ a low level protocol to exchange data between two phones. There are currently two approaches for this low level data exchange: a connection-based approach and a connectionless approach. In the former, one phone advertises a GATT service, with an app-specific service identifier, and waits for another phone to make a connection; they would then exchange data. This is typically symmetric, both phones act in both roles. In the connectionless approach, the payload is embedded in advertising/beacon data, which is then picked up by another phone scanning for this data. Unfortunately it is not currently possible for an app to reliably implement the connectionless approach without low-level operating system support (such as the upcoming Apple/Google protocol).

One of the early implementations of contact tracing is the BlueTrace/OpenTrace protocol, used in Singapore's TraceTogether app, which uses a Bluetooth GATT protocol to implement a connection-based handshake protocol between two phones. This protocol has been adopted by several other regions, including Australia (COVIDSafe), Alberta, Canada (ABTraceTogether), and Poland (ProteGO). However, many other protocols also rely on a very similar design, including the DP-3T project (used in several European countries) and the NHSX app built for the UK.

The connection-based handshake protocol works as follows:

- One phone will act a peripheral, by creating an instance of a GATT server which contains a Service identified by a UUID, e.g., in TraceTogether (Singapore) and in COVIDSafe (Australia), the UUID used is b82ab3fc-1595-4f6a-80f0-fe094cc218f9. The actual payload (the TempID) is stored as a value in an attribute (a GATT Characteristic) in that service.
- Another phone, acting in the central role, scans for nearby BLE devices, looking specifically for the service UUID for the app, and connects to the server. It then reads the TempID payload stored in the service’s characteristic.
- The client then performs a write operation to the same characteristic to upload its own TempID. It then disconnects.
- Both phones utilise private addresses, which change on a short interval (minutes), which means that the device is not traceable.
- This exchange is unencrypted, and no long-term association (i.e. pairing or bonding) is created between the devices.

Some protocols (e.g. DP-3T) attempt to rely on advertising beacons (i.e. a connectionless protocol), however, in order to support iPhones, they still provide a connection-based fallback mechanism involving data exchange using GATT.

Overview of the BLE pairing & bonding protocol

To understand how the vulnerability works, one needs to understand the pairing protocol for BLE devices. We give a very quick summary of relevant aspects of the protocol here.

The pairing protocol aims to establish a secure link between two BLE devices. This is done by using a variant of the Elliptic Curve Diffie Hellman (ECDH) protocol to establish a short term key, which is then used to exchange Long Term Keys (LTK) and Identity Resolving Key (IRK). Once these keys are stored, the devices are “bonded”. The IRK can be used subsequently to decrypt the private addresses of a paired device back to its “true” identity address. This is to allow a device to recognise its paired device, even when the private address changes. In other words, once you know a device’s IRK, you can always turn any private address back into the identity address. The IRK may change after a phone is factory-reset, but the identity address persists even after factory reset. It may be possible to spoof the Bluetooth public address in a rooted Android phone, but we are not aware of any way of doing this in an official Android OS installation.

The pairing protocol involves an initial exchange of public keys of the devices, which is prone to the man-in-the-middle (MITM) attack. To defend against MITM, the BLE pairing protocol (as of Bluetooth version 4.2) provides four methods for confirming the authenticity of the public keys; of main interests to us is the most basic method called “Just Works”. This method involves the devices sending each other nonces (random numbers) and uses them to confirm the public keys that have been exchanged. This method offers only a very weak protection against MITM (if at all), and is normally used in a setting where the user is in control of both devices. The “Just Works” method is used when the peripheral does not have input or output capabilities, so methods that require entering PIN or comparing code in devices are not possible.

The silent pairing bug

The bug is related to a feature of the Android GATT implementation, specifically in how it handles a read or a write operation from a client (BLE central role) to a server (BLE peripheral role). If an app, acting in the central role as a GATT client, issues a read/write command to a characteristic on a remote peripheral’s GATT server and receives an 'Insufficient Authentication' error, the Android Bluetooth framework will automatically initiate a pairing process.

This is an expected behaviour. In most cases, a pairing process will trigger a prompt to the user of the phone to confirm. Depending on the protocol used (e.g., “Just Works”, Numeric comparison, etc), the user may need to compare some numbers displayed on both devices, or enter a PIN code, or at least click on a pop-up to confirm. There will always be a prompt if the pairing is initiated by the peripheral.

However, in all versions of Android tested (6.0 to 10.0), if the pairing request is initiated by the central (i.e., the user's phone, even if it was initiated in response to the peripheral returning “Insufficient Authentication”), then, depending on the protocol used, there may or may not be a user confirmation involved. If the attacker can force the pairing to use the Just Works method, then no confirmation will be displayed on Android and the pairing happens silently in the background. This behaviour and how it can be exploited to perform a silent pairing has been discussed in the literature, see, e.g., [Zhang et. al. 2019, Xu et al 2019]. The mass adoption of contact tracing apps just made this exploit much easier to perform.

The Insufficient Authentication error can also be triggered in a situation where a client tries to read a protected attribute (indicated by an authentication/encryption flag). The client device will then initiate a pairing protocol before attempting to read the attribute again.

The reasoning for this behavior is likely that if the phone initiated the connection, then the remote device is implicitly trusted, and so the “Just Works” method is allowed to proceed silently. This is not normally a concern, as there is never normally a situation where phones will just connect to random devices; a connection is always either:

- Directly initiated by a user action via the system Bluetooth configuration, or in a vendor’s application (i.e. connecting a newly purchased device such as a heart rate monitor)
- Re-connecting to an already-known device.

However, these contact tracing apps violate this assumption. They constantly seek to establish connections to any other nearby device advertising the desired service, and then perform read and write operations to exchange TempIDs.

An attacker can trigger the pairing process by the following simple attack:

- The attacker sets up a GATT server, with a service with the same UUID as the one the contact tracing app is using, and sets up a characteristic with a read attribute, with authenticated and encrypted flags set.
- The attacker waits for the client app to connect to it. As soon as the client issues a read command on the authenticated/encrypted attribute, the Insufficient Authentication error is returned.
- As the client device is not already paired with the server, then an automatic bonding process will be initiated. The attacker would need to set up the server so that it indicates it does not support any input or output device, and the bonding process will be confirmed silently via the “Just Works” method.
- The attacker will now receive the IRK for the remote device.

Note that it is not necessary for the attacker to set the attribute flag for the characteristic to authenticated/encrypted. The attacker could simply respond to every read request with an Insufficient Authentication error.

Here is a video of this in action: https://youtu.be/tlWqLVfdzG0

Proof-of-concept exploit

We have developed two different proof-of-concepts (PoC) demonstrating how this can be triggered. One is based on the Linux “bluez” Bluetooth stack, the second uses the “BlueKitchen btstack” which runs on embedded devices. The source code will be provided upon request. However, anyone with moderate knowledge of Bluetooth programming should be able to write their own exploit based on the information given above.

For obvious reasons, we do not include an exploit on an actual app in the PoC, but developers of the affected apps should be able to tailor our code to test their own implementation. The exploit requires very basic Python scripts for the server side, and any off the shelf bluetooth scanner app from Android will work as a client. For this test, we use the nRF Connect Mobile app available from the Google Play App Store.

Reproducing the exploit

bluez

The test was carried out in a standard Kali distribution (but any recent Linux distro, e.g., Ubuntu 18.04 would do), running on a laptop with a built-in Bluetooth low energy device. There are two scripts included in the PoC: a bash script (setup.sh) and a python script (exploit1.py). Here are the steps to test the silent pairing issue:

1. Run the setup.sh script as root user.

   sudo ./setup.sh
   This will enable the bluetooth device and sets up appropriate parameters, in particular, the pairing protocol (using BLE protocols rather than legacy protocols), and the randomisation of bluetooth addresses. The latter allows the server to avoid blacklisting by the client app due to caching strategies implemented in the app. It would also force the pairing process to exchange IRK.

2. Launch a Bluetooth monitor application. Wireshark is great, and is already installed in Kali.

3. Run the exploit script

   sudo python3 exploit1.py

   This will advertise a GATT server that exposes a service UUID. For this PoC, we use the UUID: 12345567-1595-4f6a-80f0-fe094cc218f9. This GATT server contains a characteristic that supports two operations: encrypted read and encrypted write.

4. Test the GATT server with a BLE debugger. For our tests, we used nRF Connect Mobile. Locate the GATT service and try to read the advertised characteristic, and watch the automated pairing happen (the status will change from “not bonded” to “bonded”).

5. You can inspect the HCI log in Wireshark to see the handshakes for the pairing. Watch for the 'Insufficient Authentication' message, that is the trigger for the automated pairing.

To make the exploit work for your app, you may need to tweak the GATT service, e.g., for COVIDSafe and TraceTogether, you would need to change the 'Manufacturer Data' field to 0xff03, in order to trigger the app to perform a characteristic read.

BlueKitchen btstack

This test was carried out using a modified version of the `hid_keyboard_demo.c` and `spp_le_counter.c` demos that comes with the btstack distribution, using the libusb btstack platform with a CYW20704A2-based USB adaptor.

`cve-2020-12856.c` implements a simple BLE GATT server with a single service (defined in `cve-2020-12856.gatt`) containing a characteristic that will return “Insufficient Authentication” on first write. The LE security manager is configured to enable “Just Works” pairing.

1. Using a BLE debugger (e.g. nRF Mobile Connect) on a phone, scan for and connect to the USB adaptor.

2. The demo will print out the “random” private address of the phone.

3. Locate the GATT service and try to write to the characteristic, and watch the automated pairing happen (the status will change from “not bonded” to “bonded”).

4. The demo app will now print out the identity address and the IRK for the phone.

5. Press <enter> to enable the HID profile device. This will connect to the phone using the newly-discovered identity address.

6. The phone will indicate (via a status icon) that a keyboard is connected, and any keypresses entered into the demo app will be sent to the phone.

To make this exploit work with your contact tracing app, the advertising data and service/characteristic uuids will need to be updated.

Sample HCI log

Here are some excerpts of a log file generated on the server side in one of our bluez tests. The client was a Pixel 4 XL phone. This log was actually produced using an official release of the COVIDSafe app.

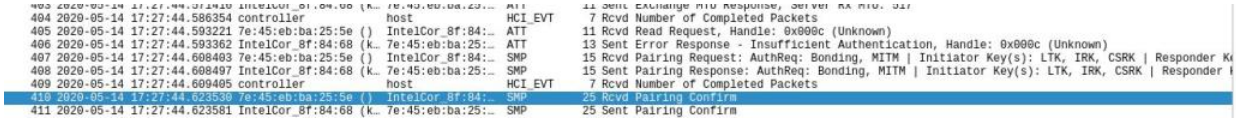

Figure 1. Insufficient Authentication error sent, followed by receipt of pairing request.

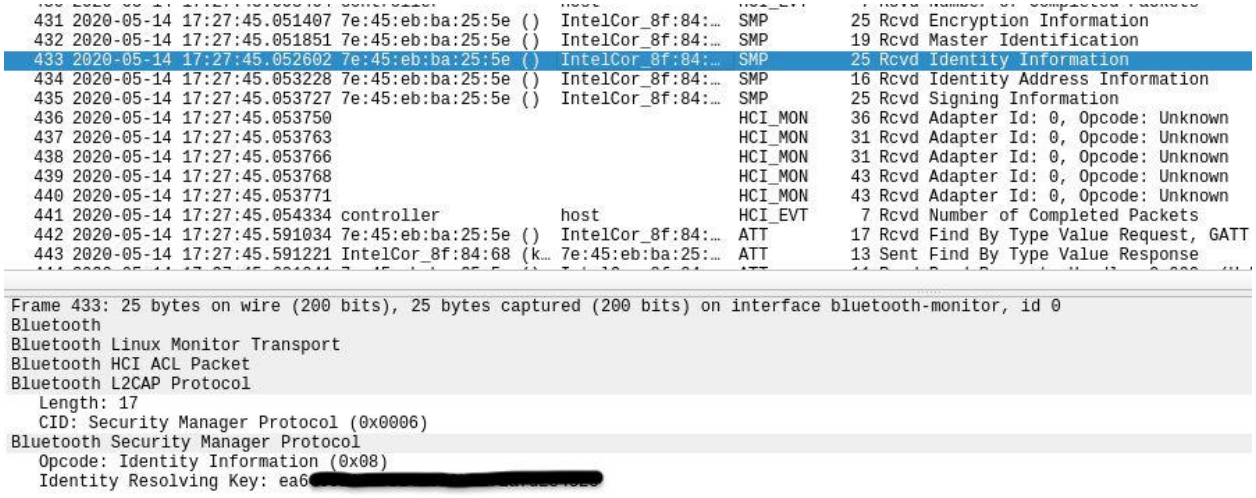

Figure 2. IRK received from the client.

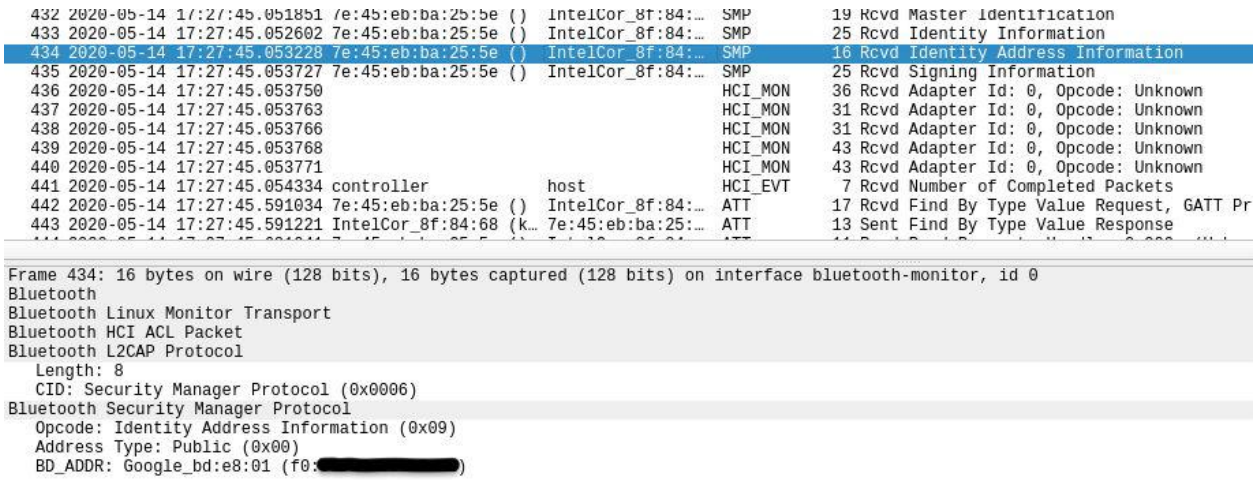

Figure 3. Client's public address received.

iPhone

When this technique is carried out against an iPhone target, the iPhone will still display a pairing prompt, which avoids this happening silently. However, there are two additional factors that still make this interesting on iOS devices.

1. The displayed prompt says "`<attacker controlled text>`" would like to pair with your iPhone. It is very easy to therefore create a very convincing prompt message (see Figure 4). If the user clicks "Pair", then the IRK is immediately available to the attacker. Additionally, while this prompt is visible, before the user clicks cancel, the device name (e.g. "<firstname>'s iPhone") is available to the attacker via the Generic Access / Device Name characteristic.
2. Most iPhones are permanently acting as BLE advertisers to support the Airdrop feature. This vastly simplifies how an attacker can track a device (compared to the L2CAP ping described below) as the attacker can passively detect the advertising beacons to see the private address.

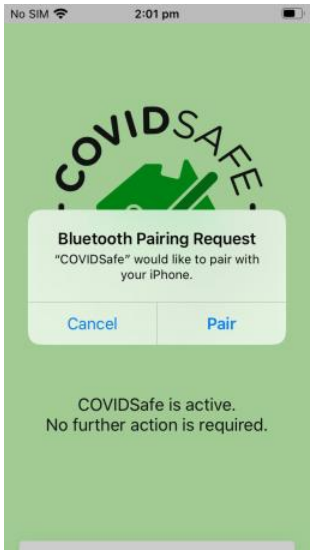

Figure 4. An example of a pairing request prompt on iPhone.

Mitigations

We believe that in order to mitigate the silent pairing issue, the affected apps will need to be redesigned to use a connectionless model, which means that the TempID payload needs to be embedded in the advertisement data. However, there are currently too many restrictions on what an iPhone app running in the background can do for it to be possible for an app to implement an effective contact tracing system while also fully supporting iPhones. This gives two main options:

- Remove support for iPhone while the app is backgrounded. While it is technically possible to make this work with the current Android APIs and (foreground) iOS APIs, it is not currently possible to customise the advertising payload on a background iPhone app. Obviously, the lack of support for background iPhones would be a huge detriment to the usability and effectiveness of the app.
- Use the Apple/Google collaboration (called the “Exposure Notification API”). However, this forces a move to the decentralised model, which is not particularly favoured by some governments and some health experts.[1]

A couple of potential mitigations have been suggested, but we believe they would be ineffective:

- Modifying the client so that it checks the flags associated with a characteristic before performing a read operation, and only performs the operation if the flags contain no encrypted or authenticated attributes. However, as described above, it’s possible for the characteristic to return the insufficient authentication status even without this flag set.

[1] See, e.g., https://www.washingtonpost.com/technology/2020/05/15/app-apple-google-virus/

- Monitor the pairing status and cancel any pairing in progress. We have not tested this mitigation, but based on the previous work by Zhang et a, 2019 it seems that the pairing state can be observed but there is no mechanism from within the app to stop the pairing process, at least not for the versions of Android they tested (Android 6 - 9). So by the time an app can detect a pairing status change, the attacker already has the IRK.

Security and Privacy implications

Linkability of TempIDs

Bluetooth address randomisation limits the attacker's ability to link TempIDs, at least without continuous tracking. Even without the silent pairing issue, continuous tracking of users may be possible, depending on how often the TempIDs are rotated. Bluetooth private addresses are typically rotated at a 15 minute interval, whereas TempIDs rotation may take longer, e.g., in COVIDSafe, the TempIDs are rotated every 2 hours. So by continuously observing the changes in the private addresses and TempIDs, the attacker may be able to link two tempIDs (sharing the same private address) or two private addresses (sharing the same TempID). But once the attacker obtains the IRK, the attacker will be able to determine whether two TempIDs obtained in two (non-continuous) encounters belong to the same user, assuming the attacker logs the (private) Bluetooth addresses during those encounters as well as the tempIDs.

Re-identification and permanent tracking of devices

Once the IRK (and therefore the identity address) of a device is known, the major concern is that it is now possible for an attacker to re-identify that device.

Even if the owner of the device notices that the pairing has been created and removes it, if the attacker runs the same process again, it will receive the same IRK. Additionally, the attacker already knows the identity address, which will never change, even after a factory reset.

As long as Bluetooth is enabled, devices will respond to L2CAP pings on their identity address, even when no other Bluetooth functionality is in use. So knowing the identity address is sufficient to know whether a given target is currently in Bluetooth range, allowing an attacker to detect the presence of the target device even after the contact tracing app is uninstalled and no other Bluetooth functionality is currently in use.

```
$ sudo l2ping aa:bb:cc:dd:ee:ff
Ping: aa:bb:cc:dd:ee:ff from 00:11:22:33:44:55 (data size 44) ...
44 bytes from aa:bb:cc:dd:ee:ff id 0 time 7.36ms
...
```

BlueFrag / CVE-2020-0022

There is also a well-known Android issue “BlueFrag” (CVE-2020-0022) [Ruge 2020], affecting all versions of Android earlier than 10.0 (approximately 84% of the market). This issue hasn’t been patched in earlier versions, likely because it requires the attacker knowing the device’s identity address, which previously wasn’t possible.

At best, CVE-2020-0022 is a denial-of-service vector, allowing an attacker to trivially crash the system Bluetooth service, which requires the user to manually re-enable Bluetooth. We have demonstrated this on phones running Android 7.0 to 8.1. However, crashes were not observed consistently in our tests on newer devices running Android 9 to 10. Some devices like Samsung S20 (Android 10) and Huawei P30 (Android 9) seem to be unaffected, whereas Google Pixel 4 XL (Android 10, 5 Oct 2019 patch) is affected. Tests done by Ruge showed it worked for some devices running Android 9 as well.

However, CVE-2020-0022 also allows an attacker to leak data from the device, and in some cases allows for remote code execution (RCE) in the context of the system Bluetooth service. This is a much more targeted attack, but has extremely serious consequences. The memory leak attack (using the fancy_leak.py script from [Ruge 2020]) appears to work on both old devices (e.g., Nexus 6P) and newer devices we tested, running Android 9 (Huawei P30, Oppo Reno2 Z). Figure 5 shows an example of a memory leak from Nexus 6P, triggered by this script, showing the location of some installed apps in the phone. Repeating the attacks multiple times may allow the attacker to leak information from different memory locations.

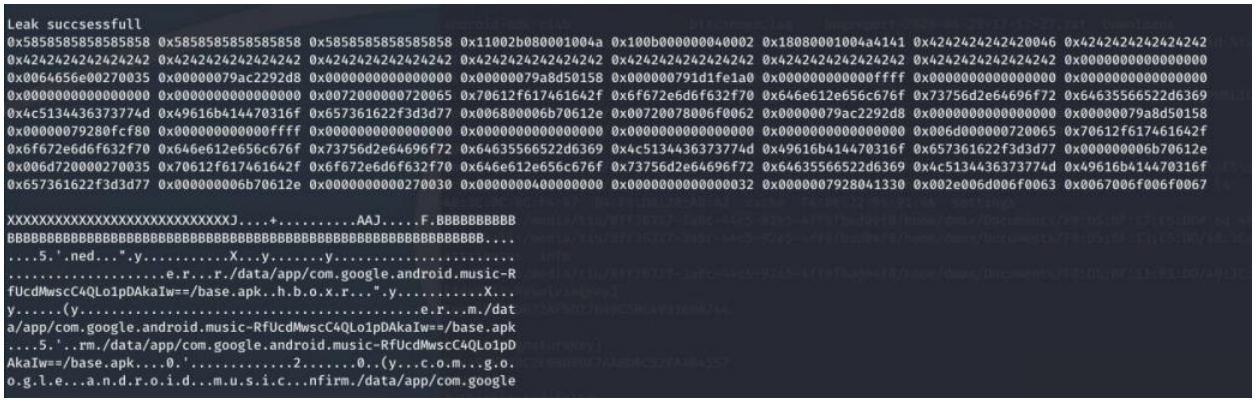

Figure 5. Example of memory leak in Nexus 6P, using CVE-2020-0022 exploit.

Profile switching

With the identity address and an existing (silent) BLE bonding in place, an attacker can now attempt to connect as a different type of device (including non-BLE device types using the “classic” Bluetooth profiles). Our experiments show that while Android will always confirm before

sharing contacts or phone call history, it will automatically and silently enable access to functionality requested by devices such as input mode (keyboard and mouse devices), or media (audio sink, i.e. headphones/speakers).

We have a practical demonstration of this that performs the following steps against an Android 8.1 target:

- The attacker's device advertises a BLE GATT Service that looks like the contact tracing service.
- The contact tracing app on the target Android device connects to this service and attempts to read and write from the characteristic, to which the attacker returns "Insufficient Authentication" (as above). This triggers the silent pairing and bonding.
- The attacker now has the identity address of the target device.
- The attacker now enables the Bluetooth HID profile, and initiates a connection to the target.
- The target then sees that it is already bonded with this device and silently performs Bluetooth "classic" pairing, generating a BR Link Key, then allowing the keyboard to be enabled.
- The attacker can now remotely control the target device.

We have also demonstrated this with other Bluetooth profiles:

- OBEX: Allowing the attacker to send a file to the target (although the pairing is silent a prompt is shown to accept the file transfer). (Android 9.0 target)
- AVRCP / A2DP: Allowing the attacker to appear as a headset, controlling media playback and receiving audio data. (Android 8.1 target)

Android 10.0 appears to have fixed this behaviour. Although the initial BLE pairing is still silent, the re-pairing for the classic profile will trigger a prompt. However, the prompt still contains attacker-controlled text, allowing for a relatively convincing prompt that users might still click on (see Figure 6).

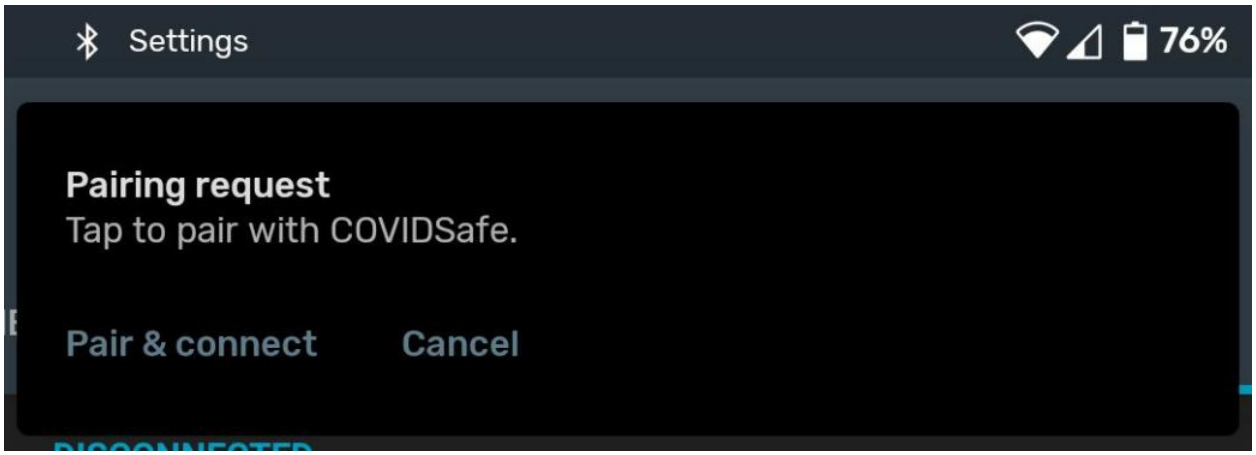

Figure 6. Android 10 prompt

REFERENCES

[Zhang et. al. 2019] Yue Zhang, Jian Weng, Rajib Deyγ, Yier Jin, Zhiqiang Lin, and Xinwen Fuγ. On the (In)security of Bluetooth Low Energy One-Way Secure Connections Only Mode. Technical Report.

[Xu et. al. 2019] F. Xu, W. Diao, Z. Li, J. Chen, K. Zhang. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. NDSS 2019.

[Ruge 2020] Jan Ruge. CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE – BlueFrag. April 2020.

APPENDIX: Analyses of affected apps

Australia's COVIDSafe

Status: confirmed vulnerable

We have tested the exploit on the COVIDSafe app (version 1.0.17) on a range of devices with different Android versions. In all cases, we managed to obtain the IRKs of the devices and to successfully pair with the devices without any user interaction.

- Samsung S20 (Android 10, 1 March 2020 patch)
- Google Pixel 2 (Android 10, 5 May 2020 patch)
- Google Pixel 2 XL (Android 10, 5 May 2020 patch)
- Google Pixel 4 XL (Android 10, 5 October 2019 patch)
- Huawei P30 (Android 9, 1 August 2019 patch)
- Oppo Reno2 Z (Android 9, 5 September 2019 patch)
- Nexus 6p (Android 8.1, 5 December 2018 patch)
- Nexus 5X (Android 8.1.0, 5 December 2018 patch)
- Samsung Note 5 (Android 7, 1 August 2018 patch)

This indicates that the issue affects all recent major Android versions and major vendors of Android phones.

TraceTogether/OpenTrace

Status: likely vulnerable

TraceTogether and COVIDSafe share many similarities -- in particular in the protocol for exchanging tempID. See, for example, the GATT server code for OpenTrace (on which TraceTogether is based on) and COVIDSafe. We have not tested the exploit on TraceTogether app, but we are confident that it is similarly affected.

ABTraceTogeher

Status: potentially vulnerable

We have not examined the source code (it is not yet available at the time of writing) of ABTraceTogether or tested the app, but based on a third-party research on the decompiled code, it seems that it is derived from OpenTrace, so it will share similar issues with COVIDSafe and TraceTogether.

UK NHS Contact Tracing App

Status: potentially vulnerable

The following is based on a quick analysis of the source code at https://github.com/nhsx/COVID-19-app-Android-BETA

The BLE related code shows a setup for a GATT server, with a handler for a characteristic read: app/src/main/java/uk/nhs/nhsx/sonar/android/app/ble/GattServer.kt

```
private val service: BluetoothGattService =
    BluetoothGattService(SONAR_SERVICE_UUID, SERVICE_TYPE_PRIMARY)
        .also {
            it.addCharacteristic(
                identityCharacteristic
            )
            it.addCharacteristic(
                keepAliveCharacteristic
            )
        }
```

```
fun start(coroutineScope: CoroutineScope) {
    Timber.d("Bluetooth Gatt start")
    val callback = object : BluetoothGattServerCallback() {
        override fun onCharacteristicReadRequest(
            device: BluetoothDevice,
            requestId: Int,
            offset: Int,
            characteristic: BluetoothGattCharacteristic
        ) {
            gattWrapper?.respondToCharacteristicRead(device, requestId, characteristic)
        }
```

The scanner code also contains an explicit call to read a characteristic:

app/src/main/java/uk/nhs/nhsx/sonar/android/app/ble/Scanner.kt

```
private fun read(
    connection: RxBleConnection,
    txPower: Int,
    scope: CoroutineScope
): Single<Event> =
    Single.zip(
        connection.readCharacteristic(SONAR_IDENTITY_CHARACTERISTIC_UUID),
        connection.readRssi(),
        BiFunction<ByteArray, Int, Event> { characteristicValue, rssi ->
            Event(characteristicValue, rssi, txPower, scope, currentTimestampProvider())
        }
    )
```

So an attack similar to that used for COVIDSafe app can be launched here; by advertising a service with the expected UUID (SONAR_SERVICE_UUID), with a characteristic with UUID as in SONAR_IDENTITY_CHARACTERISTIC_UUID specified in the build configuration.

DP-3T

Status: potentially vulnerable

We have not performed any actual tests on DP-3T implementations. The following analysis is based on a quick code review. We scan for any indications that a GATT connection is made and a characteristic is read.

DP-3T uses a shorter form of tempIDs, and embeds the payload (tempID) in the advertisement data, and in most cases the tempIDs are exchanged using a connectionless method. The only exception seems to be when the server is an iPhone app running in the background. In this case, the client app will need to connect to the server and explicitly read the characteristic

value. From code analysis, it seems that the client app will try to connect to all Apple devices to find the service UUID. This is indicated in a comment in BleClient.java:
dp3t-sdk/sdk/src/main/java/org/dppppt/android/sdk/internal/gatt/BleClient.java

```
// Scan for Apple devices as iOS does not advertise service uuid when in background,
// but instead pushes it to the "overflow" area (manufacturer data). For now let's
// connect to all Apple devices until we find the algorithm used to put the service uuid
// into the manufacturer data
scanFilters.add(new ScanFilter.Builder()
                .setManufacturerData(0x004c, new byte[0])
                .build());
```

This can then be exploited in a similar way as in the case with COVIDSafe. The attacker would need to impersonate an iOS device, e.g., setting up an advertisement that resembles a backgrounded service UUID, waits for a read request, and then issues an Insufficient Authentication error to force a silent pairing.

The characteristic read operation can be found in the GattConnectionTask class:
dp3t-sdk/sdk/src/main/java/org/dppppt/android/sdk/internal/gatt/GattConnectionTask.java

```
@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        BluetoothGattService service = gatt.getService(BleServer.SERVICE_UUID);

        if (service == null) {
                Logger.d(TAG, "No GATT service for " + BleServer.SERVICE_UUID + " found, status=" + status);
                finish();
                return;
        }

        Logger.i(TAG, "Service " + service.getUuid() + " found");

        BluetoothGattCharacteristic characteristic = service.getCharacteristic(BleServer.TOTP_CHARACTERISTIC_UUID);

        boolean initiatedRead = gatt.readCharacteristic(characteristic);
        if (!initiatedRead) {
                Logger.e(TAG, "Failed to initiate characteristic read");
        } else {
                Logger.i(TAG, "Read initiated");
        }
}
```

```
@Override
public void onCharacteristicRead(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status) {
        Logger.i(TAG, "onCharacteristicRead [status:" + status + "] " + characteristic.getUuid() + ": " +
                        Arrays.toString(characteristic.getValue()));

        if (characteristic.getUuid().equals(BleServer.TOTP_CHARACTERISTIC_UUID)) {
                if (status == BluetoothGatt.GATT_SUCCESS) {
                        if (characteristic.getValue().length == EPHID_LENGTH) {
                                callback.onEphIdRead(new EphId(characteristic.getValue()), gatt.getDevice());
                                scanResult.getRssi();
                        } else {
                                Logger.e(TAG, "got wrong sized ephid " + characteristic.getValue().length);
                        }
                } else {
                        Logger.w(TAG, "Failed to read characteristic. Status: " + status);
                        // TODO error
                }
        }
        finish();
        Logger.d(TAG, "Closed Gatt Connection");
}
```

COVID WATCH

Status: potentially vulnerable

We have not tested or examined the source code, but we note the following specification given in [their github repository]():

> *To overcome the above limitations, the protocol uses both broadcast-oriented and connection-oriented BLE modes to share TCNs. The terminology used for BLE devices in these modes are:*
>
> - *Broadcaster and observer in broadcast-oriented mode.*
> - *Peripheral and central in connection-oriented mode.*
>
> *In both modes, the protocol uses the 0xC019 16-bit UUID for the service identifier.*
>
> *In broadcast-oriented mode, a broadcaster advertises a 16-byte TCN using the service data field (0x16 GAP) of the advertisement data. The observer reads the TCN from this field.*
>
> *In connection-oriented mode, the peripheral adds a primary service whose UUID is 0xC019 to the GATT database and advertises it. The service exposes a readable and writeable characteristic whose UUID is D61F4F27-3D6B-4B04-9E46-C9D2EA617F62 for sharing TCNs. After sharing a TCN, the centrals disconnect from the peripherals.*

This sounds like a similar method adopted in DP-3T to handle backgrounded iOS app, which is done by resorting to a connection-based method. Therefore we believe it is similarly vulnerable to the silent pairing attack.