A Remote Crash Exploit on the COVIDSafe App v2.0 (Android)

Author: Alwen Tiu
Created: 2020-12-21
Last updated: 2021-02-14

Summary of the problem

COVIDSafe version 2.0 for Android has an unsafe dereferencing of an object, leading to a runtime null pointer exception, crashing the app. The crash can be triggered by writing a specific payload to a Bluetooth GATT characteristic of COVIDSafe, that is used to exchange data related to contact tracing. An attacker within the bluetooth range of the target phone running COVIDSafe 2.0 will be able to crash the app, causing it to be inactivated for a period of time. In some cases the app can be unavailable for 30 minutes, but in other situations, it can be unavailable until the user restarts the app manually, or restarts the phone.

A quick summary of the exploitation method: To crash the COVIDSafe app, connect to the GATT server on the phone running the app, and write the payload 0x010100XX where XX can be any byte. Here's a link to a video demonstrating the attack. The payload may need to be written twice (varying the last byte, e.g., for the first write use 00 and second write use 01) to ensure the app crashes.

Disclosure

I discovered this vulnerability on December 21st, 2020 and notified DTA on the same day. This vulnerability was fixed in Covidsafe 2.1, released on Feb 2nd, 2021.

Technical analysis

In the following analysis, unless otherwise stated, we assume the Android version of the COVIDSafe app.

Prior to COVIDSafe 2.0, phones running the COVIDSafe app exchange information through a GATT characteristic with UUID B82AB3FC-1595-4F6A-80F0-FE094CC218F9 that is both readable and writable. Two phones A and B can exchange information in two ways: either A reads B's payload from B's characteristic, and writes its own payload to B's characteristic (so B acts as a server and A a client), or the other way around (A acts as the server and B the client). In COVIDSafe 2.0, two new characteristics (among others) are introduced:

- **Payload characteristic**, with UUID 3e98c0f8-8f05-4829-a121-43e38f8933e7. This is a read-only characteristic.
- **Android signal characteristic**, with UUID f617b813-092e-437a-8324-e09a80821a11. This is a write-only characteristic.

The normal protocol for payload exchange uses only the payload characteristic, with two phones reading each other's payload on this characteristic.

However, in the case where a phone cannot advertise its services due to hardware limitation, it will use the signal characteristic to write its payload to a target phone, and read the payload characteristic of the target phone. It's this feature of the signal characteristic that we will exploit to remotely crash a COVIDSafe app running on a target phone.

In addition to writing payload, the signal characteristic can be used to update the RSSI information for a device, or for 'payload sharing' (currently this feature does not seem to be implemented yet).

When a remote device writes a payload to the signal characteristic of a phone, it triggers a call to the function onCharacteristicWriteRequest in the class ConcreteBLETransmitter. This function will check the data written by the remote device to determine which protocol to apply. Specifically, for the data to be interpreted as a payload, it must comply with the following format:

- The first byte must be 0x01.
- The second and the third byte specifies a 16-bit integer (in little endian), denoting the length of the payload.
- The rest of the bytes is the payload data.

So for example, the following sequence of bytes (presented as a HEX string)

010100AA

specifies a payload data of length 1 byte, whose value is the byte 0xAA.

In a normal exchange, the payload will be a UTF8 encoding of a JSON record, containing fields such as 'msg' and 'rssi'. But for this exploit, the actual format of the payload does not matter.

The call to onCharacteristicWriteRequest first creates a BLEDevice object corresponding for the remote device:

```
val targetDevice = database.device(device)
```

If this is the first time the remote device is registered, various fields of the targetDevice (in particular rssi) are not initialised, and thus may be null, as is the case with the rssi field. Then the payload is parsed to determine the type of payload data. In case the type is a 'payload' type, the following code in onCharacteristicWriteRequest is executed:

```
SignalCharacteristicDataType.payload -> {
  val payloadData = SignalCharacteristicData.decodeWritePayload(data)
```

```
            ?: // Fragmented payload data may be incomplete
            return
  logger.debug("didReceiveWrite (dataType=payload,central={},payload={})",
            targetDevice, payloadData)
  // Only receive-only Android devices write payload
  targetDevice.operatingSystem(BLEDeviceOperatingSystem.android)
  targetDevice.receiveOnly(true)
  targetDevice.payloadData(payloadData)
  onCharacteristicWriteSignalData.remove(device.address)
  if (responseNeeded) {
    server.get()?.sendResponse(device, requestId,
                BluetoothGatt.GATT_REQUEST_NOT_SUPPORTED, offset, value)
  }
  return
}
```

This updates the payload of the targetDevice and other fields. Crucially, the rssi field of targetDevice is not updated (hence remains null). The update of payload triggers a call to the function bleDatabaseDidUpdate in the class ConcreteBLESensor. In the case where the update was done in response to a 'payload' type, a call to query the device's rssi is made

final RSSI rssi = device.rssi();

This will return a null reference, since the targetDevice was a new device seen for the first time. The next line of code then tries to access the 'value' field of rssi, which triggers the null pointer exception.

final Proximity proximity = new Proximity(ProximityMeasurementUnit.RSSI, (double) rssi.value);

Exploitation

To perform the exploit, one can use a BLE scanner (e.g., nRF connect app) to identify a target (this can be determined through the service UUID and the manufacturer data in the BLE advertisement), and then connect to the target phone. The next step is then simply done by writing a 4-byte payload of the form 0x010100XX where XX can be any byte.

Note that the COVIDSafe app caches Bluetooth MAC addresses it has seen, so to trigger the crash one may need to make sure that the BLE scanner that is used to send the payload does not advertise any GATT services, to ensure COVIDSafe does not try pick up the address of the scanner during its routine scans. For example, if you run the scanner from an iPhone, the crash may not be triggered reliably.[1]

In my test, using Samsung Galaxy S20, the app crashed as soon as this payload was received. However, it was restarted almost immediately by Android ActivityManager. Performing the

[1] Thanks to Richard Nelson who pointed this out.

attack a second time crashed the app again, but this time it was not restarted until about 30 minutes later. However, performing the attack for the third time seemed to prevent the Bluetooth service from functioning properly -- there was no BLE advertisements observed from the phone and it was not attempting any connection to nearby phones.
Since the app caches the payload, every time the exploit is launched, one may need to vary the last byte to ensure the device database is updated (triggering the vulnerability).

A potential accidental crash?

If a phone running COVIDSafe app does not support GATT advertisement (e.g., Samsung J6, according to this VMWare Herald document), the app will use the signal characteristic to write its payload to a remote device. The sequence of requests to the remote device seems to be decided in the function nextTaskForDevice in the class ConcreteBLEReceiver. The current implementation schedules readPayload first, followed by writePayload and then writeRSSI. It is not clear to me whether readPayload will result in an RSSI value being assigned in the remote device. If not, when the writePayload is sent, this could have the potential of accidentally triggering the bug that crashes the app in the remote device. I have not tested this scenario yet.

Remediation

One possible remediation is to change the payload protocol to include the RSSI information. That means changing the implementation of the client that sends the write request (ConcreteBLEReceiver).

Another remediation is to put a guard to check whether rssi is null prior to dereferencing it. Or simply remove the proximity calculation when the payload type is ‘payload’, and do it in the ‘rssi’ payload type instead. This is what the Herald code does. For some reason,in COVIDSafe 2.0, the proximity calculation is done in both the ‘payload’ case and the ‘rssi’ case. This seems to be the fix done by the COVIDSafe team judging from the code snippet here.

Acknowledgement

Thanks to Richard Nelson and Jim Mussared for their comments on an earlier draft. Richard found the bug independently, and Jim found a related bug in the iOS version of COVIDSafe 2.0. Both have reported their findings separately to DTA.