Hell2CAP 0day

Written by Barak Caspi; Edited by Urit Lanzet

TL;DR

Here's a Zero-Day vulnerability we discovered in BlueSDK, a popular embedded Bluetooth stack developed by OpenSynergy and used in close to one hundred million devices.

The vulnerability was reported in a responsible disclosure form, to protect any client of the company that uses the library, and to help the company fix it. The process with OpenSynergy was fast and effective.
We truly appreciate their responsibility, professionalism and security awareness.

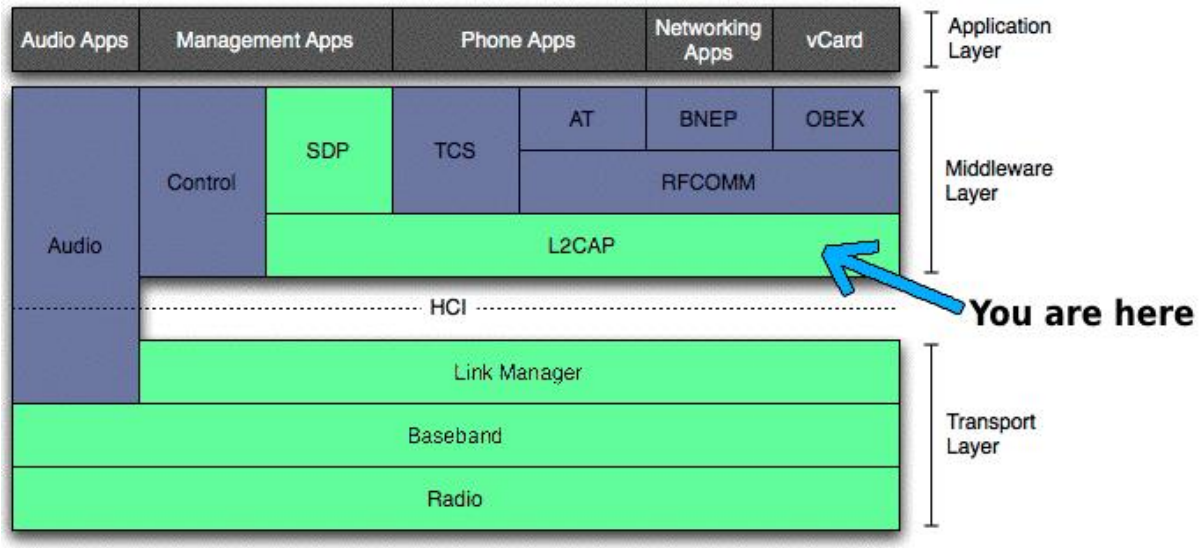
The vulnerability allows proximate unauthenticated attackers to execute arbitrary code on a device running BlueSDK as its Bluetooth stack. The core problem is a state machine bug, that can be exploited to cause an integer underflow which leads to a buffer overflow.
BlueSDK versions 3.2 through 6.0 are affected.

Disclosure Timeline

- 2018-10-08 - Vulnerability discovered
- 2018-10-23 - Proof of concept completed
- 2018-10-28 - The issue was reported to the vendor
- 2019-03-18 - Public disclosure

Bluetooth Classic (Pre-BLE) Primer

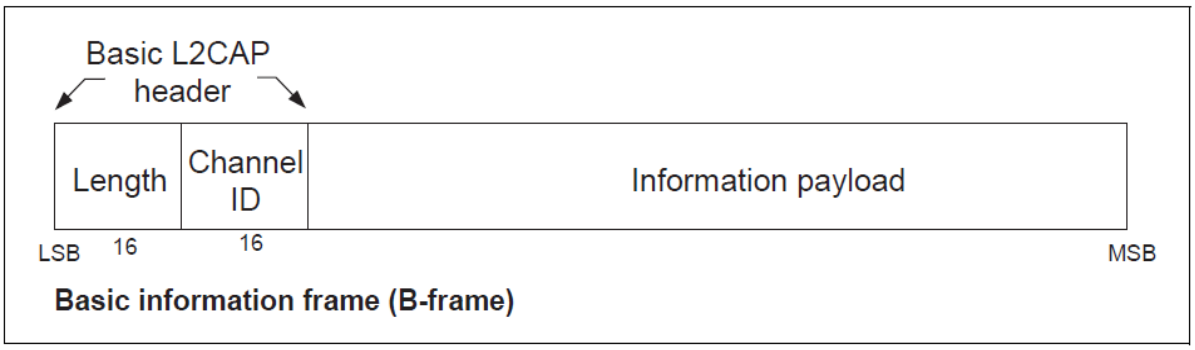
In order to understand the vulnerability, knowledge of the lower layers of Bluetooth isn't required (Radio, Baseband, LMP, HCI); it's only important to understand the interface they supply to the higher layers - specifically, establishing an ACL link (Asynchronous Connection-Less; a packet-switched type link) between two Bluetooth devices, and the ability to send data packets, upon which higher layers are encapsulated.



Bluetooth protocol stack

L2CAP is a core protocol in the Bluetooth protocol stack, encapsulated over an ACL link. It is in charge of protocol multiplexing, flow control, quality of service, segmentation & reassembly and more.
Protocol multiplexing is the most important responsibility of L2CAP. It allows creating virtual connections between logical entities on both devices, much like TCP allows two devices to communicate simultaneously in different higher-level protocols.

These connections are termed “Channels”. Each endpoint of the channel is identified by a CID (Channel ID). The multiplexing is implemented with a CID field in the packet header which acts as a "destination" field for the packet; On reception, L2CAP routes the packet to the appropriate high-level protocol which handles the specified channel.

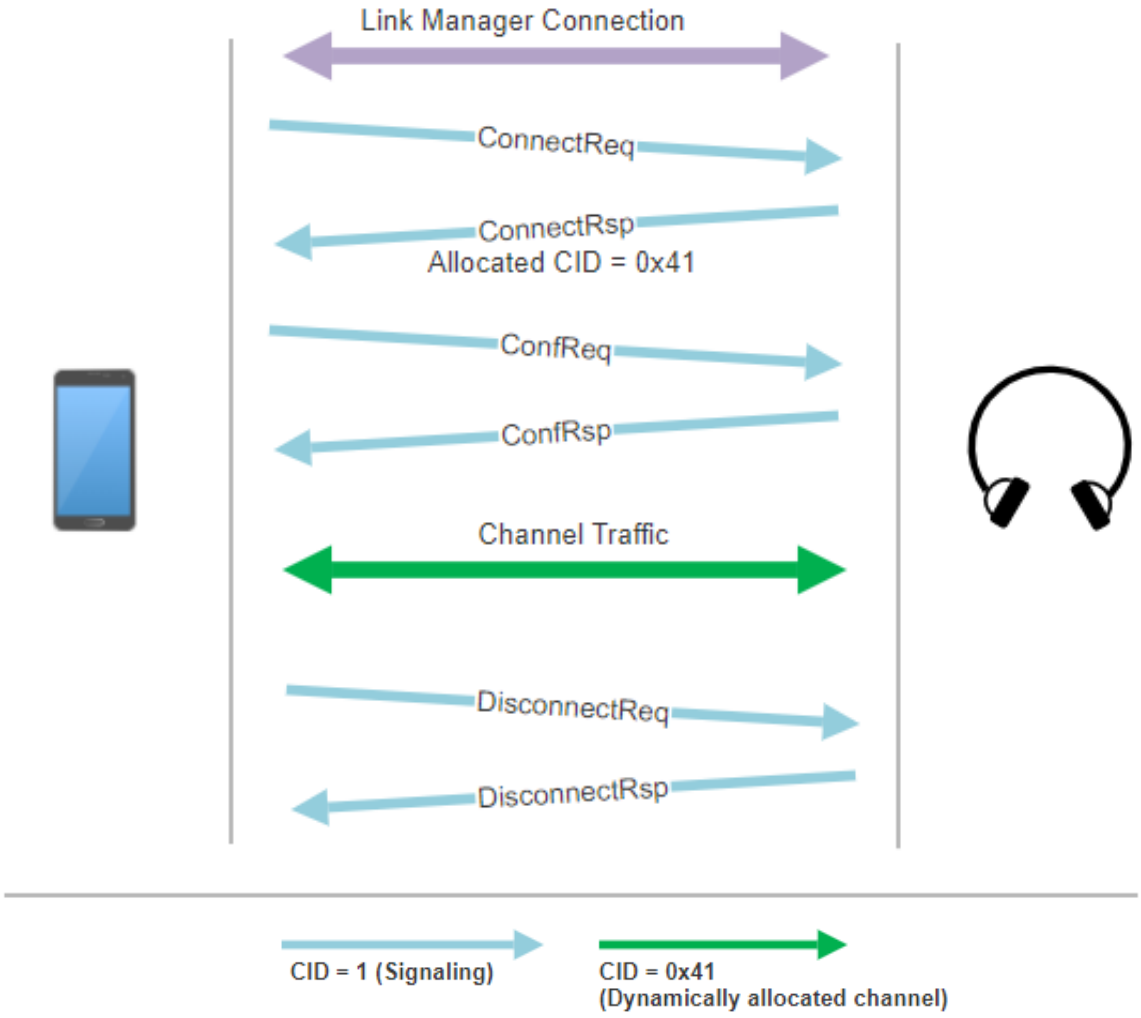


Excerpt from Bluetooth Core Spec v3.0 + HS, page 1208

L2CAP Signaling

Initially, upon establishing a link between two Bluetooth devices, there is only one L2CAP channel available for communication – the signaling channel.
Through this channel, administrative operations are performed, in a request and response fashion. For example, creating a new channel (i.e. connecting to a higher-level protocol) happens by sending a Connect command on the signaling channel, that contains an identifier of the higher-level protocol that the remote end wishes to connect to. This identifier is known as a PSM (Protocol Service Multiplexer); similar in essence to a port number in TCP.

Another example of a signaling command is a Configuration Request. The Configuration Request can configure parameters of other channels, such as timeout intervals, MTU (Maximum Transmission Unit), mode of operation, flow control parameters, etc.
Each configuration request can supply a different set of options to configure.



L2CAP Channel establishment example

It's important to understand that the configuration phase of a channel is negotiation-based. Endpoints can reply negatively to configuration requests, until the other endpoint sends a configuration request that is acceptable.
Also, many of the parameters need to be configured independently per communication direction. For example, the channel's MTU, which is configured in this way, can be different for each receiver in the communication; and as such, each receiver must configure its own MTU.

"Hell2CAP" - L2CAP Signaling State Machine Bug

As stated above, an L2CAP endpoint can configure its receiving MTU for the current channel, by sending a configuration request to the remote endpoint, notifying it of its requested receiving MTU.

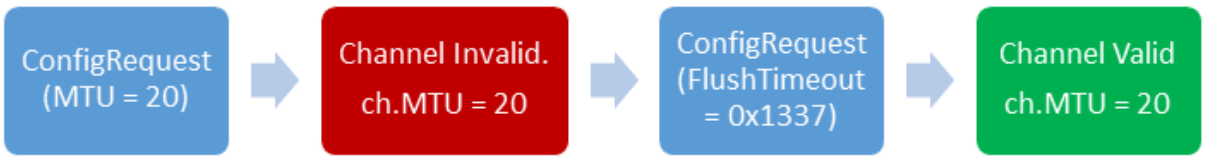
```
if ( config_option[1] == ConfigOption_MTU )
{
  in_mtu = LEtoHost16(data_iter + 2);
  channel->MTU = in_mtu;
  if ( channel->ptProtocol->minimum_protocolMTU > in_mtu )
  {
    l2cap_set_channel_as_invalid(channel);
  }
}
```

BlueSDK's code that handles the MTU option in an incoming Configuration Request (pseudo-code)

According to the Bluetooth specification, **an L2CAP channel's MTU may not be lower than 48.**
Such validation happens in the nested if - the incoming MTU is checked against the minimum allowed MTU (each protocol can supply a different MTU in BlueSDK when registering with L2CAP; the default is 48).

One can see that the incoming MTU value is stored inside the channel's structure before being checked - this is what immediately raised a red flag.
In case the incoming value is denied, the designated channel is marked as invalid, and communication cannot be performed through it – all frames will be immediately dropped by L2CAP before being routed to upper layers.

The channel still exists, though, and can be configured further. Once a valid configuration request arrives, the channel is reset to a valid state.
However, if a subsequent, valid configuration request arrives, the handling code does not take into consideration the fact that there's an invalid configured MTU.



An example for a malicious configuration flow that triggers this bug

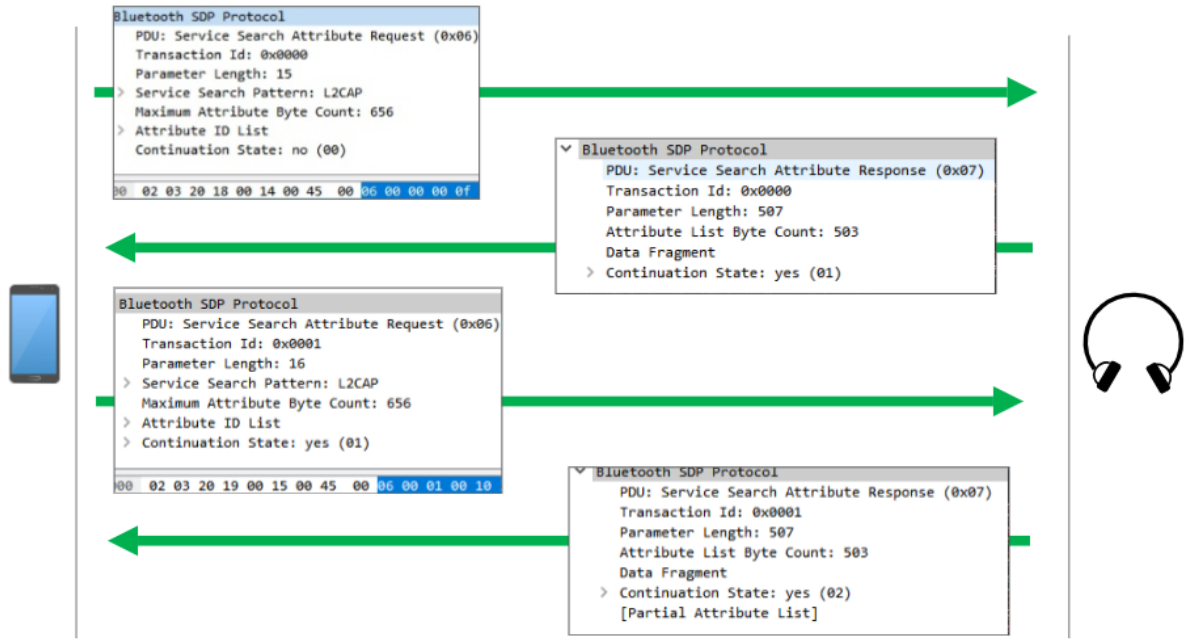
Since 0x1337 is a valid value for the FlushTimeout option, the channel is switched back to valid mode, while keeping the invalid MTU value inside the struct. Thus, **an attacker can break the constraint of the Bluetooth specification regarding the minimal MTU.**

As higher-level protocols use and trust the L2CAP layer, the vulnerability can affect execution of other protocols and cause unexpected edge conditions. After looking for targets to exploit with this vulnerability, we found the BlueSDK's SDP server code as an excellent target - its trust of L2CAP is exploitable, and it's a pre-authentication/pairing attack vector that is always available and requires no user interaction.

SDP Prerequisite Knowledge

SDP is a protocol operating above L2CAP, which allows remote devices to query the nature of the device (e.g., which services are operating, parameters regarding how the device accepts connections, etc.). As such, it is usually available pre-pairing.

SDP is a Request-Response protocol that supports fragmentation. In case a response is too large to fit into one L2CAP packet (which is determined by the current MTU), it will be sent as multiple response packets, each containing a header and a fragment of the data. Each response will contain a "cstate" (continuation state) field, which should be transmitted back to the server for subsequent fragments (each fragment has to be requested by sending the original request with the last cstate received).
The cstate field in SDP packets is defined in the specifications; however, it's defined as an arbitrary data field. The contents of it are implementation specific. In case of BlueSDK, the data is simply one byte, incrementing per request. The first request should leave the cstate empty.



Example part of SDP request-response flow with response fragmentation

The AttributeListByteCount field of the SDP request (seen in Wireshark as Maximum Attribute Byte Count) determines how many bytes the client will accept for the next response fragment.

Exploiting the SDP Server using Hell2CAP

The following pseudo-code shows how the SDP server decides the size for the current reply, which is a single fragment out of the entire response, and prepares a fragment after figuring out how long it may be:

```
1  MTU = L2CAP_GetTxMtu(_sdpInfo->CID);
2  availableSizeForFragment = (MTU - 9) & 0xFFFF;
3  ...
4  SdpStoreAttribData(_sdpInfo, _txPkt, _txPkt->bufferPtr, availableSizeForFragment);
```

BlueSDK's vulnerable SDP server code

As seen before, an SDP response fragment packet has a 9-byte header. In order to calculate the available size for the data fragment field & the cstate, the formula used is (MTU – 9). This size is passed to a function that fills the response buffer with data, up to the size passed.

The SDP code trusts the L2CAP layer to return a valid MTU; however, if the MTU is less than 9, an integer underflow will occur, that results in passing a very large size argument for the SdpStoreAttribData function, triggering a buffer overflow.

The overflown buffers (one per SDP connection) live in the global data section. Depending on which buffer is overflown, there are interesting things to overwrite in memory following the buffer.
Our exploit chooses the last buffer for overflow (by occupying all available simultaneous SDP connections), which is succeeded in memory by a PFN (function pointer variable). Overwriting the PFN means hijacking the code's flow once it is triggered.

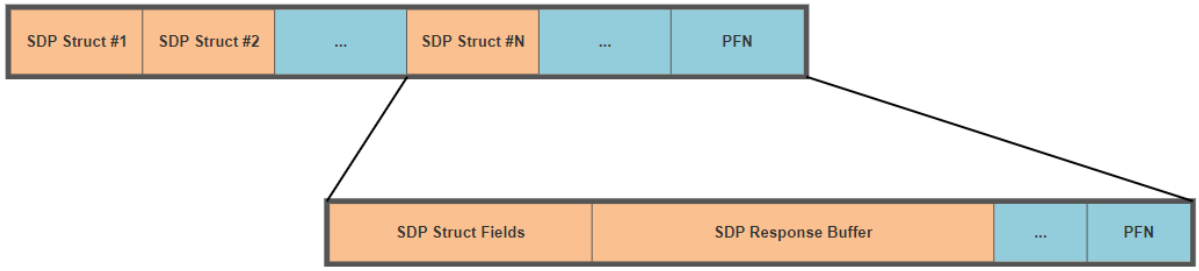


Illustration of memory we are going to corrupt

Capture the PFN

Shaping the response

Since SdpStoreAttribData overwrites the buffer with data that originates from the device itself (SDP attributes), the data is not arbitrary data chosen by the attacker. The only things that can be influenced when the packet is formed are:

a. **Which attributes are returned.**
   This is determined by the request parameters. Since the variety of available attributes is small, this approach is irrelevant.
b. **The final 1 or 2 bytes of the packet, which are the cstate.**
   Since each fragment other than the final fragment of the response increases the cstate (stored in a global variable) by 1, this value can be deduced based on previous responses' cstate.
   Moreover, it can be “primed” to specific values by a method that will be demonstrated next. This makes the final 1 or 2 bytes written in the overflow attacker-controlled.
c. **How many bytes to return per packet.**
   This is determined by the request's AttributeListByteCount field, that can be different per request packet (even between different fragments of the same search transaction).
   This also determines how “far” into the buffer will data be written, as it affects the bounds of the response buffer. This makes the offset of the final byte written in the overflow to be attacker-controlled.

Determining the "where"

The buffer can be overflown once per request-reply (i.e. per fragment). The buffer bounds are limited by the MTU, but also by another factor - the AttributeListByteCount field sent in the request.
This field determines how “far” into the buffer data will be written, as it affects the size of the response packet. It makes the offset of the final byte written in the overflow to be attacker-controlled.
By sending a AttributeListByteCount field with value X, the buffer will be overflown with X bytes (given there's enough bytes left to be transmitted in the response).

Comparison of two overflows exploiting Hell2CAP; the difference is the AttributeListByteCount sent in each request

Determining the "what"

We can write arbitrary data by performing one overflow per byte, after priming the continuation state to desired byte each time.
It’s important to notice that order of writing has to be from the highest offset to the lowest, because otherwise, consecutive writes will destroy previous bytes that have been placed carefully.

Flow for writing arbitrary sequence of bytes after the buffer

The POC was demonstrated on the target we were assessing, that uses BlueSDK and runs on ARM64, with DEP enabled.
The main module was compiled without ASLR, which simplifies the exploit.

Your PFN is belong to us now

The PFN is used as a callback which is called in a few cases. It can be triggered by an attacker with a data packet over some channel, which is also available pre-pairing.
Since DEP was enabled on our target, we couldn't just redirect the PFN to a shellcode; we had to find a way to overcome DEP.
The classic way to do this is using ROP (Return Oriented Programming), which we all know and love.
However, at no point do we overflow the stack - our primitive is a data-section buffer overflow. So, we have an intermediate task of preparing and launching a ROP.
As a result, we started looking for a stack pivot - specifically a way to overwrite the stack (since we couldn't find a gadget that relocates the stack). And since our primitive includes a call flow hijack, we decided to achieve it through JOP.

JOP (Jump Oriented Programming)

JOP is similar to ROP. Both methods revolve around executing "Gadgets", which are sequences of assembly, that when executed in the correct order perform complex tasks. The idea behind both methods is:

1. Execute an assembly instruction that is part of a larger purpose
2. Get the next gadget's address from an attacker-controlled location
3. Transfer execution to the next gadget

	ROP	JOP
Purpose	Execute Gadgets	Execute Gadgets
Method	RET (from stack)	BR <Reg/Mem> (from memory)

Example	// Gadget effect - MOV W0, W20
	MOV W0, W20
	
	// Reg restore
	LDP X19, X20, [SP,#0x10]
	
	// Next gadget taken from stack
	LDP X29, X30, [SP],#0x20
	
	// Initiate next gadget
	RET
		// Next gadget taken from memory (X0+18h)
		LDR X2, [X0,#0x18]
		
		// Gadget effect - MOV X1, X23
		MOV X1, X23
		
		// Initiate next gadget
		BLR X2

Comparison between ROP and JOP

Unfortunately, we cannot share any other exploitation details, to protect the products out there that use this library. The CVE details, once published, can be found here