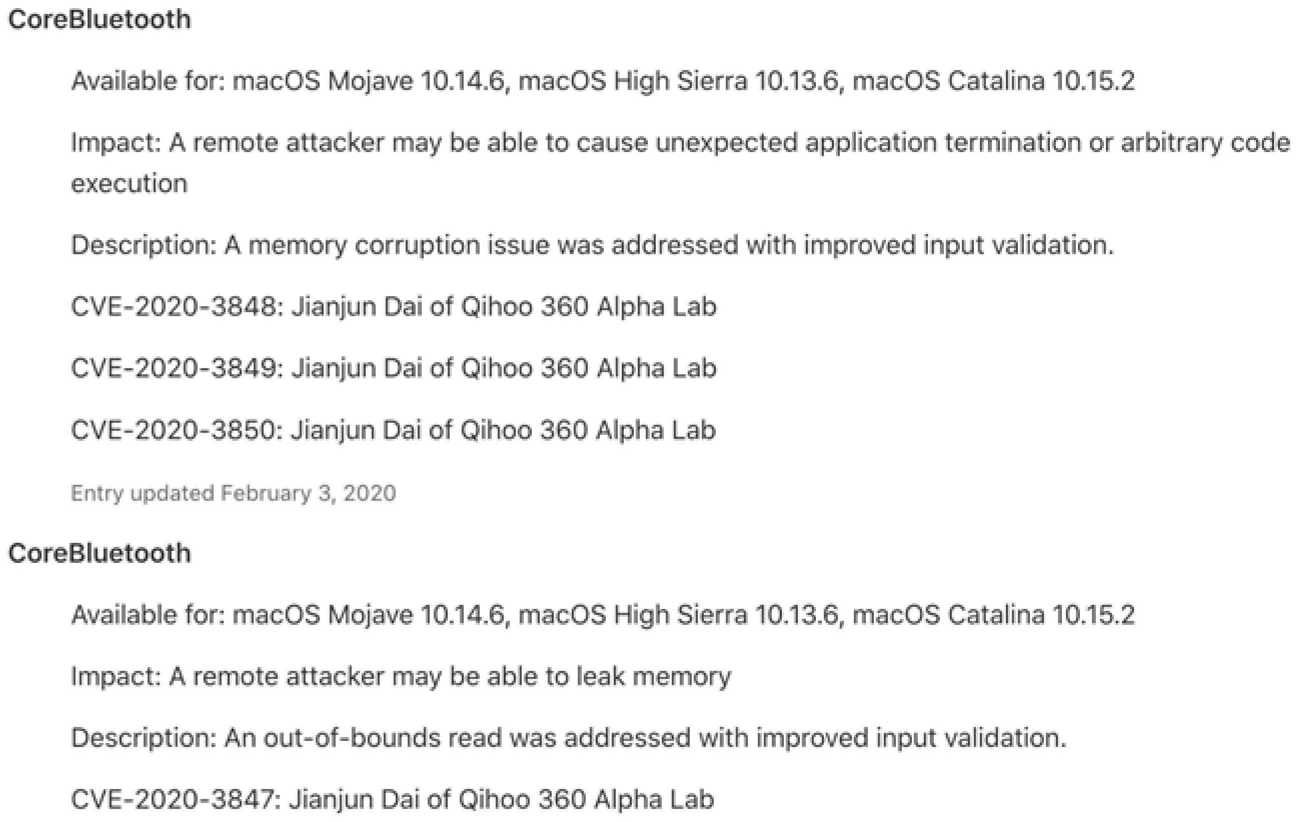
Take Down MacOS Bluetooth with Zero-click RCE

Author: Jianjun Dai(@jioundai) of 360 Alpha Lab

Zero-click vulnerabilities have become more and more popular in recent years, and the bounty for full exploit chains has also surged. In 2019, Apple Security Bounty even raised the reward up to one million US dollars for zero-click kernel code execution with persistence and kernel PAC bypass.

In the meanwhile, high prices also mean greater threat the vulnerability may pose and more difficulty to discover. Anyone who has done researches on remote zero-click vulnerabilities knows that it is very tough to get a stable zero-click exploitation as it lacks flexible interface for remote calls and has many unstable factors.

In December 2019, I submitted 5 macOS Bluetooth vulnerabilities to Apple, together with a complete report of zero-click bugs that can remotely take down macOS Bluetooth.



CoreBluetooth

Available for: macOS Mojave 10.14.6, macOS High Sierra 10.13.6, macOS Catalina 10.15.2

Impact: A remote attacker may be able to cause unexpected application termination or arbitrary code execution

Description: A memory corruption issue was addressed with improved input validation.

CVE-2020-3848: Jianjun Dai of Qihoo 360 Alpha Lab

CVE-2020-3849: Jianjun Dai of Qihoo 360 Alpha Lab

CVE-2020-3850: Jianjun Dai of Qihoo 360 Alpha Lab

Entry updated February 3, 2020

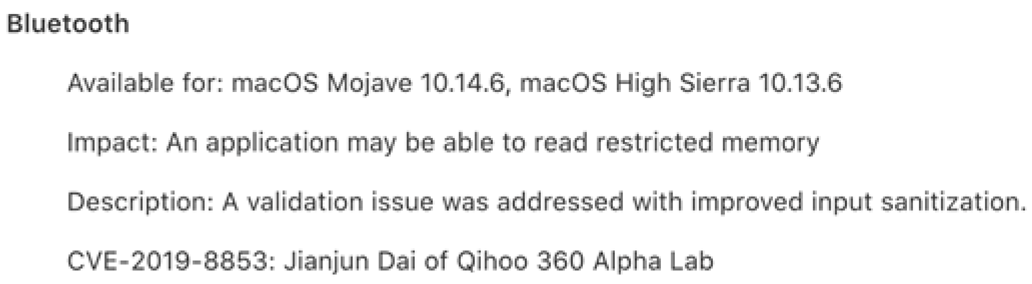
CoreBluetooth

Available for: macOS Mojave 10.14.6, macOS High Sierra 10.13.6, macOS Catalina 10.15.2

Impact: A remote attacker may be able to leak memory

Description: An out-of-bounds read was addressed with improved input validation.

CVE-2020-3847: Jianjun Dai of Qihoo 360 Alpha Lab



Bluetooth

Available for: macOS Mojave 10.14.6, macOS High Sierra 10.13.6

Impact: An application may be able to read restricted memory

Description: A validation issue was addressed with improved input sanitization.

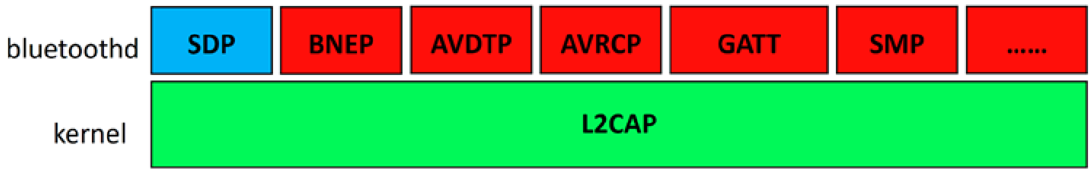
CVE-2019-8853: Jianjun Dai of Qihoo 360 Alpha Lab

The odd thing is that in the March security update, the vulnerability is numbered CVE-2019-8853 (why it’s 2019?), and macOS Catalina 10.15.3 was left out of the affected versions.

In this article, I will detail two vulnerabilities used in the exploit chain, CVE-2020-3847 and CVE-2020-3848, and how did I get the code execution. However, I will not release the exploit code itself. If you are interested, you can try to reproduce the exploit yourself.

0x0 MacOS Bluetooth

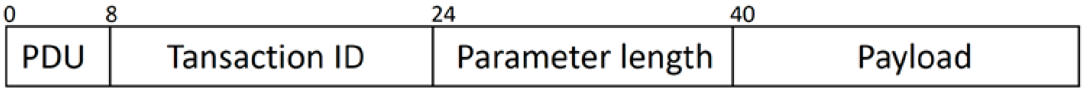
0x01 Bluetooth Architecture



In macOS, the data on the layer L2CAP (Logical Link Control and Adaptation Protocol) is processed by the kernel driver IOBluetoothFamily (the vulnerability I found in the IOBluetoothFamily was in Apple's January acknowledgement). The data on L2CAP, such as SDP, BNEP, and so on, are handled by the user mode process bluetoothd, and the bluetoothd process runs with root privilege.

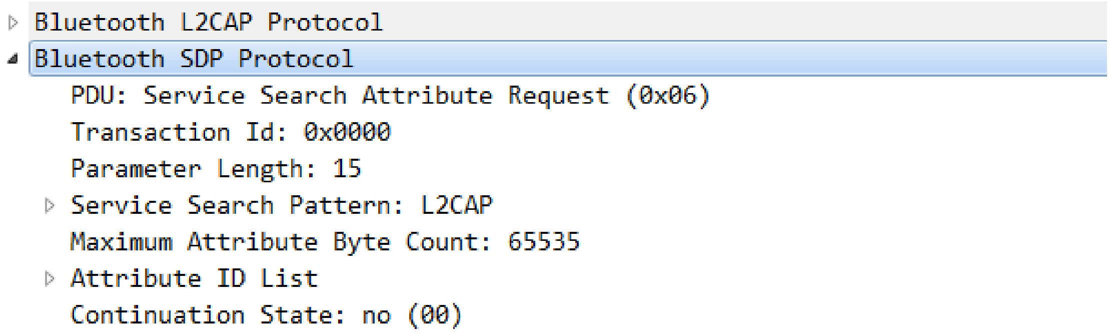
0x02 SDP Frame

The two vulnerabilities involved in this exploit are both in the processing code of SDP (Service Discovery Protocol) data frames. This section briefly introduces the SDP frame, as follows:

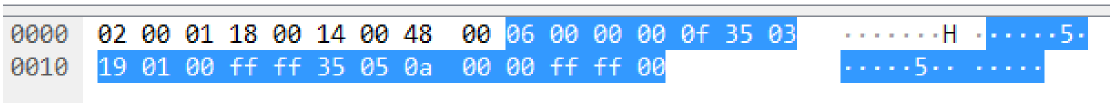


The first byte PDU field indicates the SDP request or response message. PDU = 2/4/6 indicates SDP Request, and PDU = 0/1/3/5 indicates SDP Response.

The Parameter Length field indicates the length of the payload. You can use wireshark to capture and analyze the packets as follows:



```
▷ Bluetooth L2CAP Protocol
◢ Bluetooth SDP Protocol
    PDU: Service Search Attribute Request (0x06)
    Transaction Id: 0x0000
    Parameter Length: 15
  ▷ Service Search Pattern: L2CAP
    Maximum Attribute Byte Count: 65535
  ▷ Attribute ID List
    Continuation State: no (00)
```



```
0000  02 00 01 18 00 14 00 48  00 06 00 00 00 0f 35 03   ·······H ·····5·
0010  19 01 00 ff ff 35 05 0a  00 00 ff ff 00             ·····5·· ·····
```

0x1 Vulnerability Details

0x11 CVE-2020-3847

CVE-2020-3847 can cause remote out-of-bounds read, and it exits in the (PDU=4) of function [SDPServerConnection handleServiceAttributeRequest:length:transactionID:]. To trigger the vulnerability, two SDP requets in different states should be sent.

The 1st request:

```
135  v68 = (unsigned __int64)objc_msgSend(v66, "getEncodedSize");
136  cont_state = pdata[v68 + 6];
137  if ( cont_state && cont_state != 4 )
138  {
139    objc_msgSend(pSDPServerConn, "sendErrorResponse:transactionID:", 5LL, v81);
140    v64 = 1;
141    goto LABEL_63;
142  }
143  is_cont_pkt = cont_state == 4;
144  if ( cont_state == 4 )
145  {
146    if ( !*((_QWORD *)pSDPServerConn + 7) )
147    {
148      objc_msgSend(pSDPServerConn, "sendErrorResponse:transactionID:", 5LL, v81);
149      v64 = 1;
150      goto LABEL_63;
151    }
152    v88 = *(_DWORD *)&pdata[v68 + 7];
153    cont_offset = sub_1000A3CA0(v88);
154    rem_len = *((_DWORD *)pSDPServerConn + 16) - cont_offset;
155    goto LABEL_30;
156  }
```

Line 136 cont_state reads a byte from the transmitted data packet (pdata). The first time we make cont_state = 0, so line 143 is_cont_pkt = false.

Then move onto the following lines:

```
267    v78 = (unsigned __int16)((char *)objc_msgSend(*((void **)pSDPServerConn + 1), "outgoingMTU") - 8);
268    if ( v78 < (signed int)max_list_len )
269      max_list_len = v78;
270    if ( rem_len <= max_list_len )
271    {
272      v74 = rem_len;
273      v69 = 0;
274    }
275    else
276    {
277      max_list_len -= 4;
278      v74 = max_list_len;
279      v69 = 1;
280      if ( !is_cont_pkt )
281      {
282        if ( *((_QWORD *)pSDPServerConn + 7) )
283          free(*((void **)pSDPServerConn + 7));
284        *((_DWORD *)pSDPServerConn + 16) = rem_len;
285        *((_QWORD *)pSDPServerConn + 7) = malloc(*((unsigned int *)pSDPServerConn + 16));
286      }
287    }
```

Line 270, rem_len is an indirectly controllable variable. According to the data in the request packet, query the attributes in the SDP database, Rem_len indicates the length of the query, let’s assumed it to be 0x16. max_list_len is 2-byte data read from the data packet, and it is also a directly controllable variable. We can make rem_len> max_list_len so that the code goes to the else branch.

Line 280, because is_cont_pkt = false, the final code is executed to line 284. rem_len will assign service_attr_result_len to a member variable of the object pSDPServerConn, and pSDPServerConn-> ServiceAttributeResults = malloc (0x16). The pSDPServerConn object is an object generated when an SDP socket connection is established. It is destroyed only when the connection is disconnected.

The 2nd SDP Request:

Send a second SDP request message. Make cont_state = 4, so that is_cont_pkt = true.

Line 153, cont_offset is an unsigned int variable that reads 4 bytes of data from the data packet and is also directly controllable.

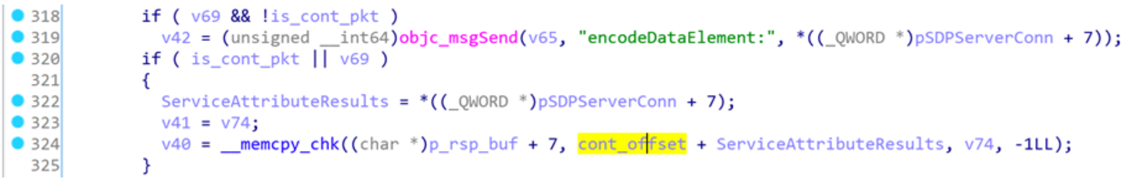
Line 154, we make cont_offset> 0x16. Assuming cont_offset = 0x18, an integer overflow occurs and rem_len = uint32_t (-2).

Line 270, rem_len> max_list_len, to enter the else branch.

Line 278, v74 = max_list_len, is also a directly controllable variable, of course, the value must be smaller than MTU (672).

280 lines, because is_cont_pkt = true, the following code will not be executed.

The code runs to the following lines:



```
318        if ( v69 && !is_cont_pkt )
319          v42 = (unsigned __int64)objc_msgSend(v65, "encodeDataElement:", *((_QWORD *)pSDPServerConn + 7));
320        if ( is_cont_pkt || v69 )
321        {
322          ServiceAttributeResults = *((_QWORD *)pSDPServerConn + 7);
323          v41 = v74;
324          v40 = __memcpy_chk((char *)p_rsp_buf + 7, cont_offset + ServiceAttributeResults, v74, -1LL);
325        }
```

Line 324, because cont_offset is controllable, it can cause an out-of-bounds read on pSDPServerConn-> ServiceAttributeResults, and the length v74 is also controllable. And p_rsp_buf will eventually be sent back to the attacker, resulting in information leakage.

0x12 CVE-2020-3848

CVE-2020-3848 can cause remote memory corruption. It exists in the function [SDPClientConnection handleServiceSearchAttributeResponse:length:transactionID:]. The code is as below:

```
103   v51 = *(_WORD *)pdata;
104   v45 = sub_100086F40(v51);
105   v44 = pdata[v45 + 2];
106   if ( v47 != v44 + v45 + 3 )
107   {
108     IOBluetoothOSLogHelper(
109       "bluetoothd",
110       2LL,
111       "-[handleServiceSearchAttributeResponse:...] Error - received packet
112       "bytes expected.\n");
113     goto LABEL_30;
114   }
115   v42 = *(_QWORD *)((char *)pSDPClientConn + 78) != 0LL;
116   v43 = v44 != 0;
```

read from packet

Line 105, v44 is a byte of data read from the data packet, assuming v44 = 0xff.

Line 116, then v43 = true.

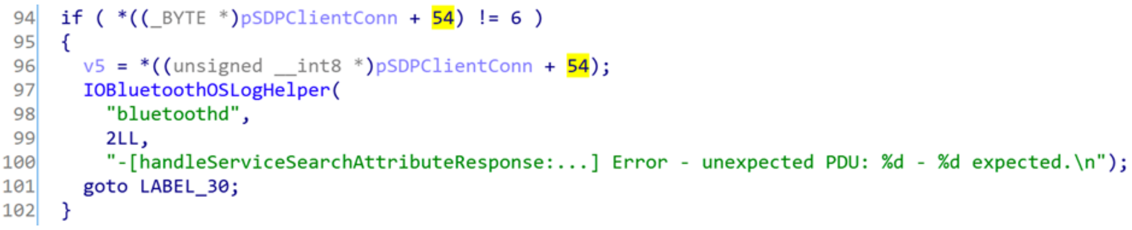
Then look at the following code snippet:

```
134   if ( v43 )
135   {
136     v37 = 0;
137     v36 = -21846;
138     v52 = (unsigned __int64)objc_msgSend(pSDPClientConn, "getNewTransactionID");
139     *(_WORD *)(*(_QWORD *)((char *)pSDPClientConn + 62) + 1LL) = sub_100086F40(v52);
140     v53 = *(_WORD *)(*(_QWORD *)((char *)pSDPClientConn + 62) + 3LL);
141     v36 = sub_100086F40(v53);
142     if ( v42 )
143       v37 = *(_BYTE *)(*(_QWORD *)((char *)pSDPClientConn + 62) + *((unsigned __int16 *)pSDPClientConn + 35) - 17);
144     v54 = *(_WORD *)(*(_QWORD *)((char *)pSDPClientConn + 62) + 3LL);
145     v55 = v44 + (unsigned __int16)sub_100086F40(v54) - v37;
146     *(_WORD *)(*(_QWORD *)((char *)pSDPClientConn + 62) + 3LL) = sub_100086F40(v55);
147     __memcpy_chk(
148       *((unsigned __int16 *)pSDPClientConn + 35) - 17 + *(_QWORD *)((char *)pSDPClientConn + 62),
149       &pdata[v45 + 2],
150       v44 + 1,
151       -1LL);
```

Line 149, * ((unsigned __int16 *) pSDPClientConn + 35)-17 is pSDPClientConn-> req_buf, and req_buf = malloc (0x20) points to a fixed length of memory. Because v44 = 0xff, memcpy caused a heap overflow, and the overflowed data was completely controllable.

0x2 Unique Features Make Perfect Zero-click

When I discovered the above memory corruption vulnerability, I was actually very frustrated because of the code in the function [SDPClientConnection handleServiceSearchAttributeResponse:length:transactionID:



```
if ( *((_BYTE *)pSDPClientConn + 54) != 6 )
{
  v5 = *((unsigned __int8 *)pSDPClientConn + 54);
  IOBluetoothOSLogHelper(
    "bluetoothd",
    2LL,
    "-[handleServiceSearchAttributeResponse:...] Error - unexpected PDU: %d - %d expected.\n");
  goto LABEL_30;
}
```

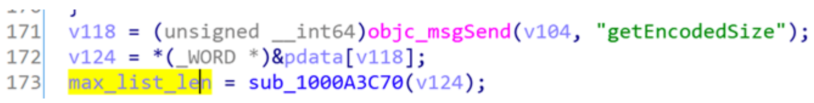
This code is to check whether pSDPClientConn has sent the corresponding Request message. If no request has been sent before, it is considered that an abnormal response message is received and the function is exited, so as not to trigger the above memory corruption vulnerability.

If you have researched the Bluetooth SDP protocol, you should know how great this check is. Many Bluetooth protocols don't do this.

Because of this code, I once considered giving up on this vulnerability. Because to trigger the vulnerability, a Bluetooth pairing connection needs to be established so that macOS can actively send SDP requests. On the surface, it is one-click, but its influence is greatly reduced. This is not the result I wanted.

The experience I have accumulated while researching Android Bluetooth vulnerabilities has helped me. I know that many manufacturers will design unique features on the Bluetooth connection of their own products to achieve some special functions.

So I decided to analyze it again in depth. Finally, hard work pays off. I found a very interesting feature in the [SDPServerConnection handleServiceSearchAttributeRequest: length: transactionID:] function:



```
v118 = (unsigned __int64)objc_msgSend(v104, "getEncodedSize");
v124 = *(_WORD *)&pdata[v118];
max_list_len = sub_1000A3C70(v124);
```

```
212  if ( max_list_len == 0xFD2D )
213  {
214    v11 = objc_msgSend(*((void **)v123 + 1), "device");
215    v85 = (void *)objc_retainAutoreleasedReturnValue(v11);
216    v12 = ((id (__cdecl *)(InquiryManager_meta *, SEL))objc_msgSend)(
217            (InquiryManager_meta *)&OBJC_CLASS___InquiryManager,
218            "defaultManager");
219    v84 = "uuid16:";
220    v83 = objc_retainAutoreleasedReturnValue(v12);
221    v82 = v83;
222    v13 = objc_msgSend(&OBJC_CLASS___IOBluetoothSDPUUID, "uuid16:", 4098LL);
223    v132 = objc_retainAutoreleasedReturnValue(v13);
224    v81 = &v132;
225    v80 = v132;
226    v14 = objc_msgSend(&OBJC_CLASS___NSArray, "arrayWithObjects:count:", &v132, 1LL);
227    v79 = objc_retainAutoreleasedReturnValue(v14);
228    v78 = (unsigned __int64)objc_msgSend(v85, "performSDPQuery:uuids:", v83, v79);
229    objc_release(v79);
230    objc_release(v80);
231    objc_release(v83);
232    objc_release(v85);
233  }
```

Line 173, max_list_len is 2-byte data read from the SDP request message;

Line 212, if max_list_len == 0xFD2D, the performSDPQuery function will be called. We can guess from the function name that it will send an SDP request. My analysis confirmed my previous guess.

In this way, I found a way to trigger a memory corruption vulnerability with zero click. I was thrilled to find this feature!

0x3 The Exploitation

A more conventional idea to a heap overflow exploitation is to heap feng shui. But when I wrote the exploitation, I encountered the following two difficulties:

- The SDP channel could not find a suitable interface to generate a large number of new objects and reside in memory.
- Other conventional channels cannot achieve zero-click, such as BNEP, GATT and other protocols. When the connection is established, a popup prompts

连接请求来自：4

连接请求来自：4

若要与设备连接，请点按“连接”以完成连接过程。

若要拒绝与设备连接，请点按“取消”。

Translation of the popup window:

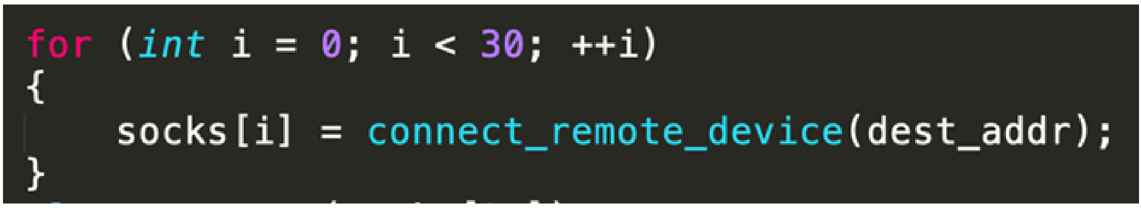
The connection is from: 4

If you want to permit the connection, please press “connect”.

Refuse please press “cancel”.

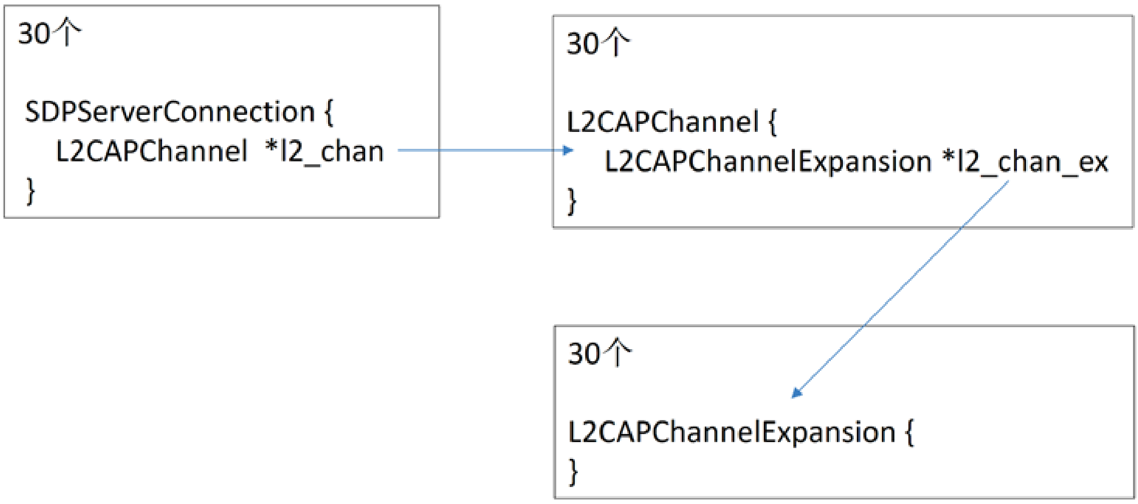
So I can only complete all the exploits through the SDP channel.

After testing and research, it is found that the same client can establish 30 SDP socket connections with macOS Bluetooth at the same time, so that 30 SDPServerConnection objects can be created.

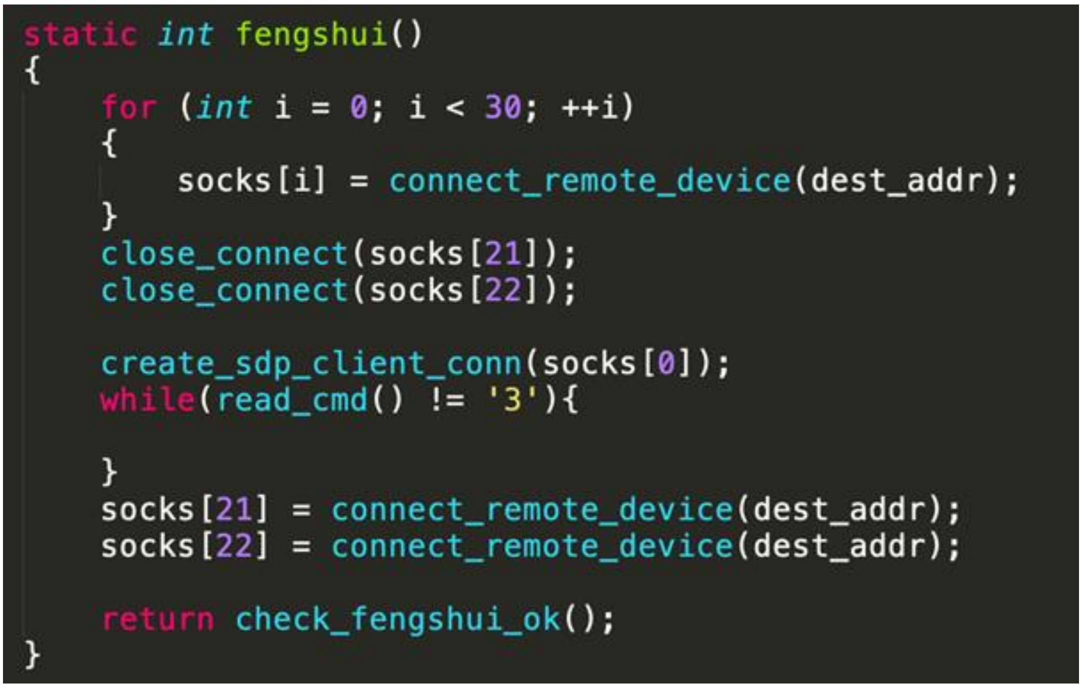


```
for (int i = 0; i < 30; ++i)
{
    socks[i] = connect_remote_device(dest_addr);
}
```

And other objects will be created in the SDPServerConnection object. In the end, I found that I can get the following relationship diagram:



Thus, 90 available objects can be laid out in memory, so that a simple heap feng shui can be completed. As for the macOS heap management mechanism, it will not be introduced here.



```
static int fengshui()
{
    for (int i = 0; i < 30; ++i)
    {
        socks[i] = connect_remote_device(dest_addr);
    }
    close_connect(socks[21]);
    close_connect(socks[22]);

    create_sdp_client_conn(socks[0]);
    while(read_cmd() != '3'){

    }
    socks[21] = connect_remote_device(dest_addr);
    socks[22] = connect_remote_device(dest_addr);

    return check_fengshui_ok();
}
```

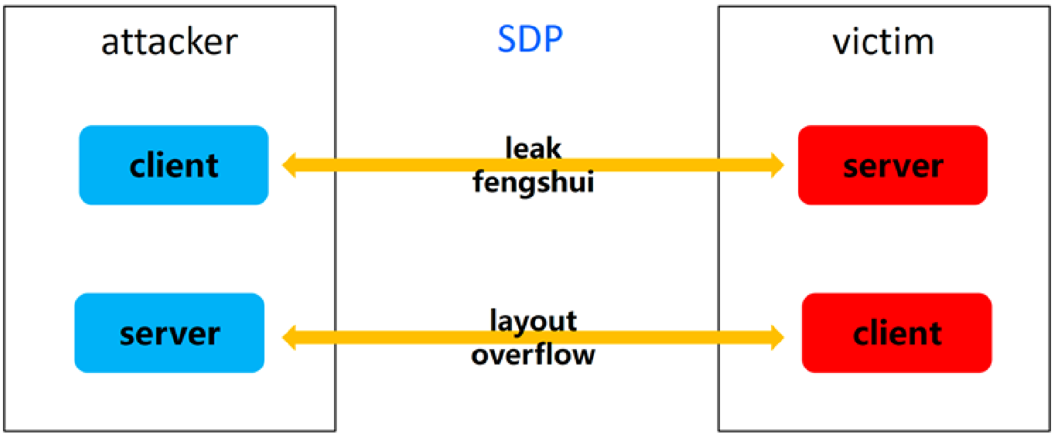
The complete exploit idea is as follows:

1. Create 30 SDP socket connections to complete the simple heap feng shui.
2. Use 30 SDPServerConn for information leakage, leak the object SDPClientConn, and obtain the address of SDPClientConn-> req_buf, and the address of SDPClientConn-> result_buf (for the memory layout later, such as fake_obj etc.)

3. Leak object SDPServerConn, find SDPServerConn objects that meet the following conditions: addr (SDPServerConn-> ServiceAttributeResults) <addr (SDPClientConn-> req_buf); keep a record of: offset = addr (SDPClientConn-> req_buf) --addr (SDPServerConn-> ServiceAttributeResults); sock [i];
4. Use sock [i] and offset to directly leak the data (<255) after SDPClientCon-> req_buf, and verify whether it is one of the following three objects: SDPServerConn L2CAPChannel L2CAPChannelExpansion
5. If a known object is successfully laid out after req_buf, a memory corruption vulnerability is triggered, covering obj-> isa, and use Objective-c's exploit techniques to complete code execution.

Otherwise, exit and return to step 1.

During the entire process of triggering and exploiting the vulnerability, the attacker's device must act as both a client and a server, as shown in the following figure:



0x4 Summary

After researching the Bluetooth protocol of Apple devices, it is known that most protocols cannot complete zero-click Bluetooth socket connection. This article mainly introduces how to find the SDP protocol vulnerabilities, explore the possibility of zero-click, and finally complete the exploitation in such a narrow gap. The article analyzes the vulnerability in detail, introduces some interesting features in the design of the macOS Bluetooth, and take advantage of them to complete the interaction-less vulnerability exploitation, and also shares ideas behind it.

Timeline

- December 1, 2019, submitted 5 vulnerabilities and an exploit report to Apple
- December 4, 2019, Apple officially confirmed the vulnerabilities
- January 29, 2020, Apple Security Update released 4 patches
- March 25, 2020, Apple Security Update released the fifth patch