The road to iOS sandbox escape

Rani Idan

@raniXCH

Summary

Apple’s sandbox may seem the “safest”, We decided to research interesting and not well known IPC. Among the history of iOS vulnerabilities, many vulnerabilities were discovered mostly on XPC, we decided to reveal the mach messages mechanism Apple still uses and poorly designed daemons based on mach message IPC.

With all of this in mind, we started to research all the mach ports accessible from within the sandbox and it revealed a new world to explore.

In order to have better understanding on the different mach message handlers, we created several research tools we are willing to share with the community. Those scripts were the key and the breakthrough to better reveal the backend of most of Apple’s API between the sandbox and the daemons. Nevertheless, we will share several vulnerabilities that were found during the research, mainly focus on the vulnerability that leads to execution of arbitrary code on most of the daemons outside the sandbox, for example, sharingd, coreduetd, SpringBoard, mDNSResponder, aggregated, wifid, Preferences, CommCenter, iaptransportd, findmydeviced, routined, UserEventAgent, carkitd, mediaserverd, bluetoothd and so on.

The vulnerability is giving full control on PC and on several registers on the vulnerable daemons and exists on all of Apple mobile devices (iOS, WatchOS and tvOS).

Moreover, We will cover possible exploitation and reveal necessary gadgets that may be used for full chain.

The white paper will cover three main goals

- Research
- Exploit
- Vulnerabilities advisory

Enumerating attack surface

In order to understand the mach servers that are accessible from within app context it is possible to use sbtool in order to easily get the sandbox profile of an app. It is possible to use sbtool for different sandbox capabilities such as files, mach and so on.

Jonathan Levin the author of sbtool shared further details in Hack in the sandbox.

Using the accessible mach servers, I targeted both com.apple.coremedia.decompressionsession (mediaserverd) and com.apple.server.bluetooth (bluetoothd).

When a mach message is being sent to one of the daemons it is being handled by different callbacks distinguished by the message id of the message.

mediaserverd

The VTDecompression API is implemented by using mach messages with the data need to be decoded to mediaserverd that will eventually send it to decoding.
One of the mach message handlers in mediaserverd is XDecodeFrame - that is the backend of the API call VTDecompressionSessionDecodeFrame.

When an app use the VTDecompressionSessionDecodeFrame API call one of the arguments is the actual buffer needed to decode, that buffer is being serialized and sent to mediaserverd. I statically analyzed the serialization process and decided it is effective to generate corrupted serialized buffers on interesting offset on the serialized buffer struct. That way we are avoiding some of the API checks on the side of the app and sending raw buffers directly to mediaserverd.

The results of generating corrupted serialized buffers and sending directly to mediaserverd were astonishing, in that way we proved it will work.
In order to profile the crashes I wrote lldb script that will send the malicious buffers to mediaserverd and on the side of mediaserverd will dump all of mediaserverd crashes with the troubled buffer.

bluetoothd

On bluetoothd I manually analyzed and understood the design of BTSession, how is the session is being created and identified by bluetoothd.

After briefly going over 132 callbacks I found one SP control (CVE-2018-4095) and one full control over PC (CVE-2018-4087) on all of the clients of bluetoothd (bluetoothd, mediaserverd, routined, sharingd, SpringBoard, and so on).
More details on both of the CVE are publicly available below with PoC code for callback vulnerability (CVE-2018-4087)

Zimperium zLabs Security Advisory

Privilege escalation/code execution in bluetoothd clients

ZIMPERIUM, INC. - ZLABS TEAM
RANI IDAN <RANI@ZIMPERIUM.COM>

Affected component:
bluetoothd (formerly BTServer) on iOS

Vendor:
Apple, Inc.

Latest vulnerable version:
iOS 11.1 (15B150) and iOS 11.2 (15C5107a) beta

Disclosure Timeline:
Bug discovered: Nov. 10, 2017
Vendor notified: Nov. 14, 2017

Summary
Session hijacking of all bluetoothd clients and adding callback to an arbitrary address.

Impact
Out-of-sandbox code execution and privilege escalation from within the default application sandbox.

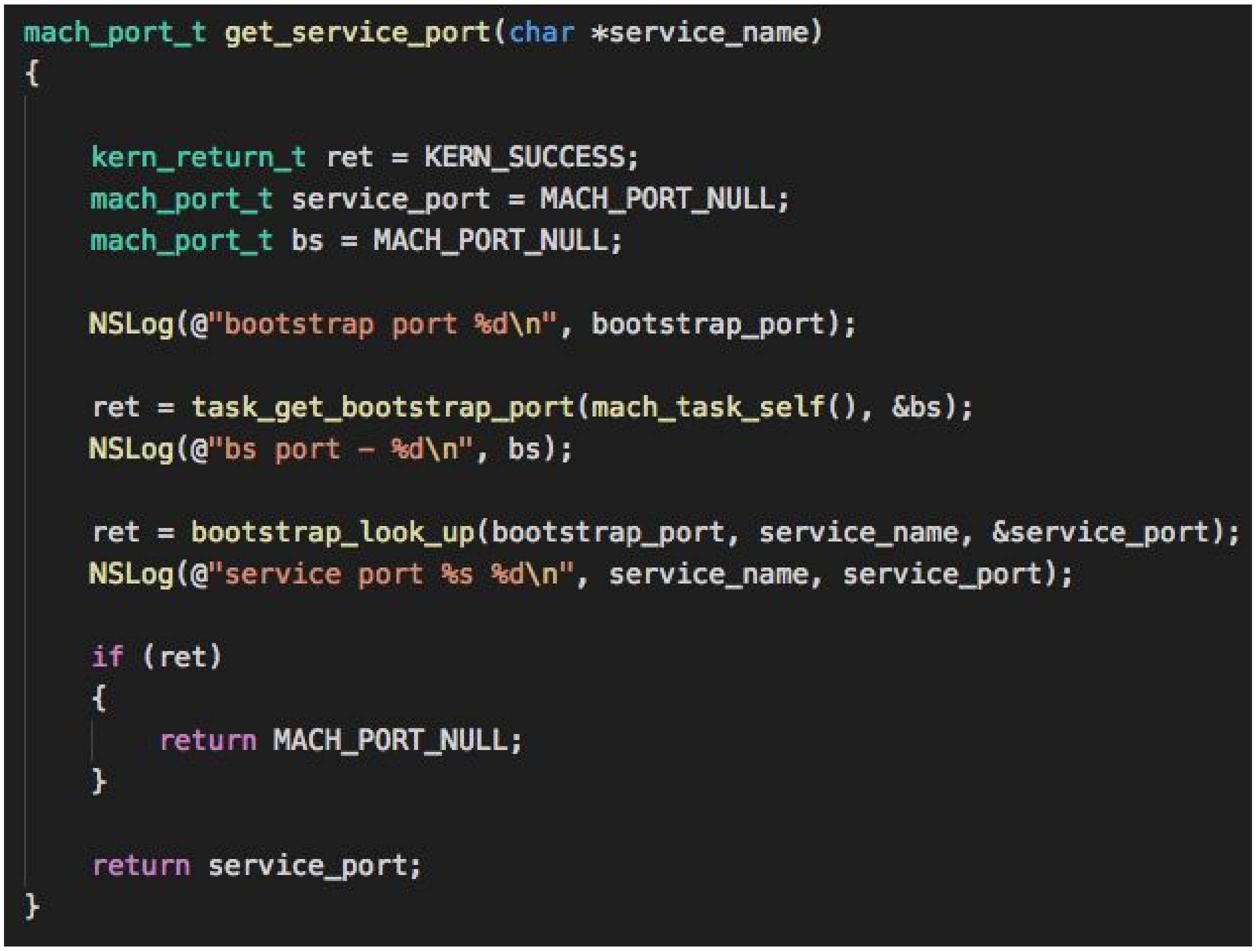
Details

The given examples and assembly snippets were taken from iOS version 11.1.1 from an iPhone 7 Plus device (“iPhone9,3”) with the MD5 sum f55c33fa856acab3424901c369e5eeb0 for the binary /usr/sbin/bluetoothd, but are also relevant to other devices and previous iOS versions as well.

bluetoothd is a service on iOS which exposes a bluetooth API to applications via mach messages.
One of the mach servers on bluetoothd is “com.apple.server.bluetooth” and its interface provides 132 functions.

The vulnerability disclosed is located in the `BTLocalDeviceAddCallbacks` MIG server handler in bluetoothd. It leads to code execution in all of the bluetoothd clients (bluetoothd, mediaserverd, routined, sharingd, SpringBoard, and so on).
The same “session hijacking” can disclose information on the client state.

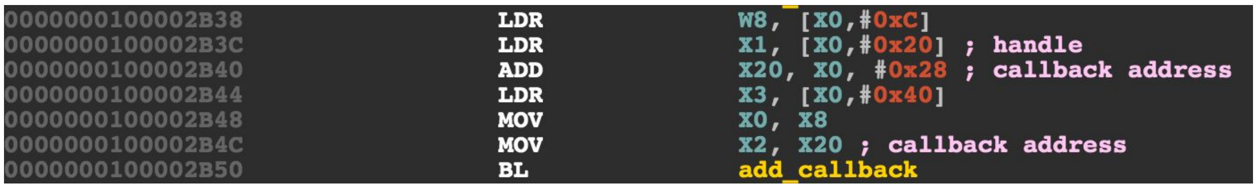
Mach messages can be sent directly to bluetoothd (renamed from BTServer in previous iOS versions) by acquiring the service (com.apple.server.bluetooth) port using `bootstrap_look_up()`.



```
mach_port_t get_service_port(char *service_name)
{

    kern_return_t ret = KERN_SUCCESS;
    mach_port_t service_port = MACH_PORT_NULL;
    mach_port_t bs = MACH_PORT_NULL;

    NSLog(@"bootstrap port %d\n", bootstrap_port);

    ret = task_get_bootstrap_port(mach_task_self(), &bs);
    NSLog(@"bs port - %d\n", bs);

    ret = bootstrap_look_up(bootstrap_port, service_name, &service_port);
    NSLog(@"service port %s %d\n", service_name, service_port);

    if (ret)
    {
        return MACH_PORT_NULL;
    }

    return service_port;
}
```

Figure 1 - acquiring the service port

The vulnerability is located in `BTLocalDeviceAddCallbacks` (handler address: 0x100002B14) and affects all of the bluetoothd clients.
The handler does check `msgh_size`, `msgh_bits` and `handle id` (offset: 0x20).

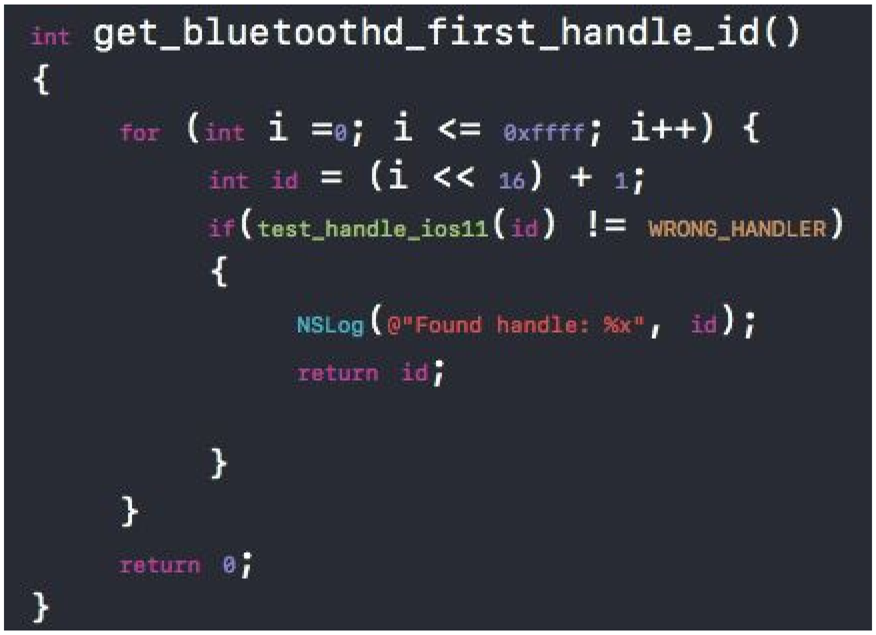


```
0000000100002B38        LDR     W8, [X0,#0xC]
0000000100002B3C        LDR     X1, [X0,#0x20] ; handle
0000000100002B40        ADD     X20, X0, #0x28 ; callback address
0000000100002B44        LDR     X3, [X0,#0x40]
0000000100002B48        MOV     X0, X8
0000000100002B4C        MOV     X2, X20 ; callback address
0000000100002B50        BL      add_callback
```

Figure 2 - The inner call to add_callback ; X0 - The mach message header

Later on, the address (offset: 0x28) will be added as a callback to the client using the client's handle id (offset: 0x20).

An attacker can gain the handle id by brute forcing because the handle id consists of 2 bytes as in figure 3.



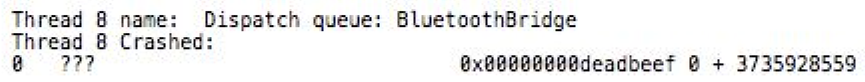
```
int get_bluetoothd_first_handle_id()
{
    for (int i =0; i <= 0xffff; i++) {
        int id = (i << 16) + 1;
        if(test_handle_ios11(id) != WRONG_HANDLER)
        {
            NSLog(@"Found handle: %x", id);
            return id;

        }
    }
    return 0;
}
```

Figure 3 - Iterating all the numbers from 0x00 - 0xffff, the function return the first handle id it found

After the attacker retrieved a legitimate handle id, this id can be used to send another mach message on behalf of the client. That will lead to adding a callback to the client with an address controlled by the attacker.

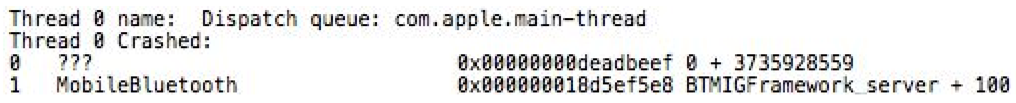
This means an attacker can hijack the session of different clients that use bluetoothd and change the code flow.

PoC is attached to the email - The code will iterate on all legitimate handle id of clients and will jump to the address as in figure 4-7.



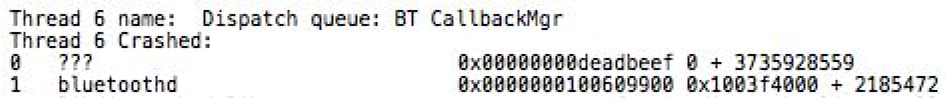
```
Thread 8 name:  Dispatch queue: BluetoothBridge
Thread 8 Crashed:
0   ???                         0x00000000deadbeef 0 + 3735928559
```

Figure 4 - Example crash from mediaserverd



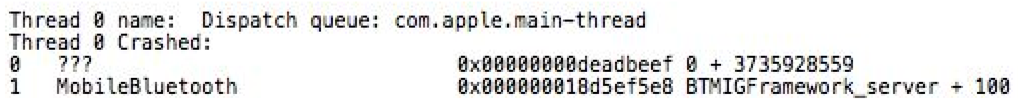
```
Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   ???                         0x00000000deadbeef 0 + 3735928559
1   MobileBluetooth             0x000000018d5ef5e8 BTMIGFramework_server + 100
```

Figure 5 - Example crash from mDNSResponder



```
Thread 6 name:  Dispatch queue: BT CallbackMgr
Thread 6 Crashed:
0   ???                         0x00000000deadbeef 0 + 3735928559
1   bluetoothd                  0x0000000100609900 0x1003f4000 + 2185472
```

Figure 6 - Example crash from bluetoothd



```
Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   ???                         0x00000000deadbeef 0 + 3735928559
1   MobileBluetooth             0x000000018d5ef5e8 BTMIGFramework_server + 100
```

Figure 7 - Example crash from coreduetd

This vulnerability can be fully exploited with any information disclosure on any process that is accessing bluetoothd via the aforementioned API.

Solution

Verify the handle id by checking the source of the mach messages using the mach message trailer.

This bug is subject to a 90 day disclosure deadline, after which the bug report will be publicized.

Zimperium zLabs Security Advisory

Memory corruption in bluetoothd

ZIMPERIUM, INC. - ZLABS TEAM
RANI IDAN <RANI@ZIMPERIUM.COM>

Affected component:
iOS

Vendor:
Apple, Inc.

Latest vulnerable version:
iOS 11.1 (15B93)

Disclosure Timeline:
Bug discovered: Nov. 03, 2017
Vendor notified: Nov. 07, 2017

Summary
Stack control on bluetoothd from sandboxed application that may lead to code execution or info leak.

Impact
Privilege escalation from within the sandbox

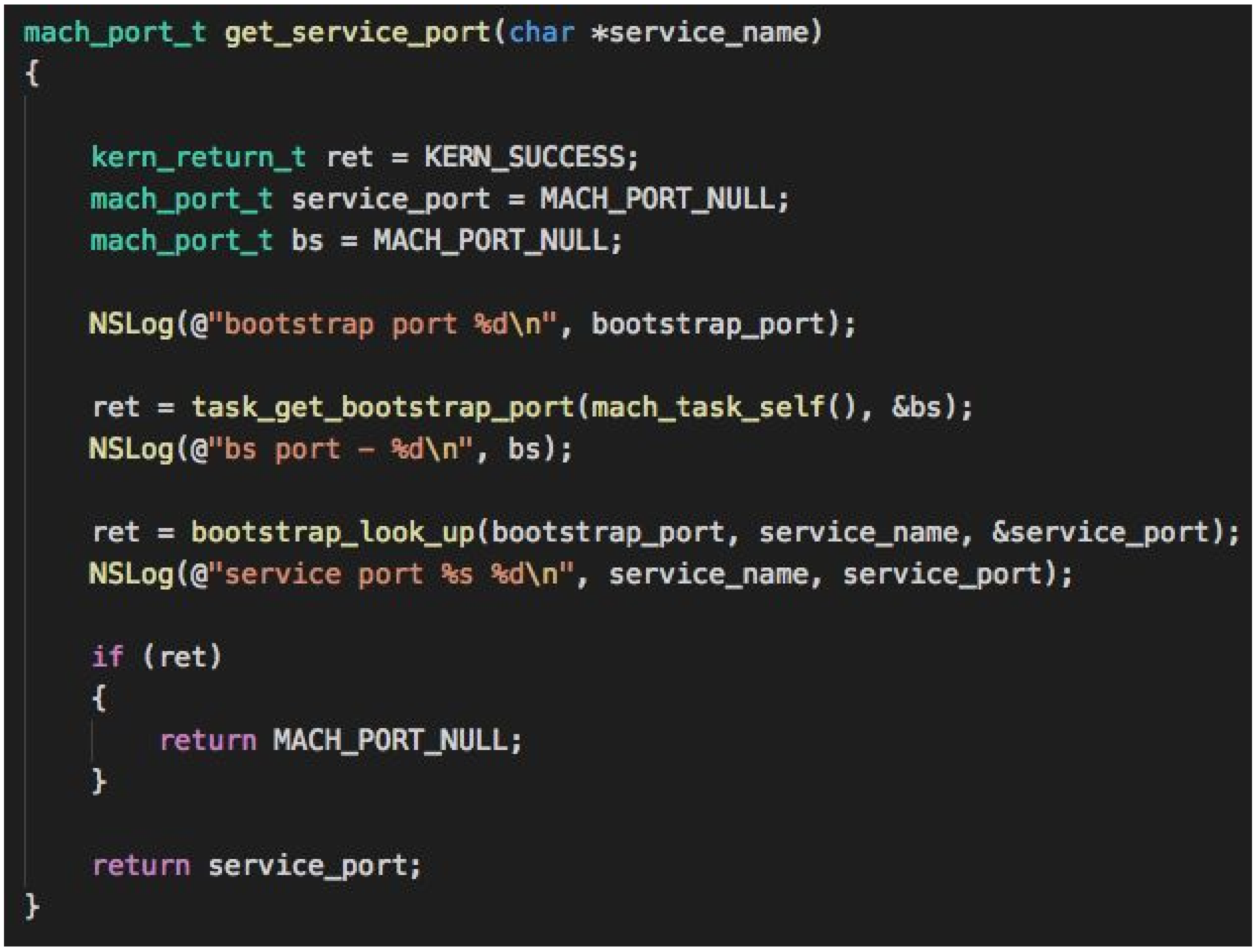
Details

The given examples and assembly snippets were taken from iOS version 11.1 (111.1.8) from an iPhone 7 Plus device (“iPhone9,3”) with the MD5\SHA1 sum f55c33fa856acab3424901c369e5eeb0 for the binary /usr/sbin/bluetoothd, but are also relevant to other devices and previous iOS versions as well.

bluetoothd is a service on iOS which exposes bluetooth API to applications via mach messages.
One of the mach servers on bluetoothd is “com.apple.server.bluetooth” which exposes 132 functions.

The bug disclosed is in several of the functions within the server (full list is in the table below), the bug leads to memory corruption by changing the stack pointer relatively by input from sandboxed application.

Mach messages can be sent directly to bluetoothd (Renamed from BTServer) by acquiring the service (com.apple.server.bluetooth) port using bootstrap_look_up.



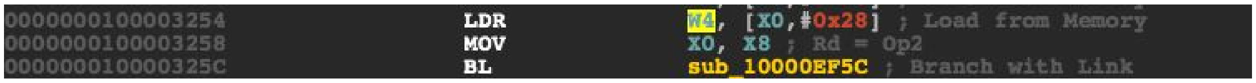
```
mach_port_t get_service_port(char *service_name)
{

    kern_return_t ret = KERN_SUCCESS;
    mach_port_t service_port = MACH_PORT_NULL;
    mach_port_t bs = MACH_PORT_NULL;

    NSLog(@"bootstrap port %d\n", bootstrap_port);

    ret = task_get_bootstrap_port(mach_task_self(), &bs);
    NSLog(@"bs port - %d\n", bs);

    ret = bootstrap_look_up(bootstrap_port, service_name, &service_port);
    NSLog(@"service port %s %d\n", service_name, service_port);

    if (ret)
    {
        return MACH_PORT_NULL;
    }

    return service_port;
}
```

Figure 1 - acquiring the service port

The bug reproduces on different mach message handlers in com.apple.server.bluetooth:

mach_msg_id	Name	Handler address (Absolute address to bluetoothd)
16	_BTLocalDeviceGetPairedDevices	0x100003210
18	_BTLocalDeviceGetConnectedDevices	0x100003354
19	_BTLocalDeviceGetConnectingDevices	0x100003410
46	_BTLocalDeviceGetDeviceNamesThatMayBeBlacklisted	0x1000043B8
52	_BTDiscoveryAgentGetDevices	0x1000046C8
107	_BTAccessoryManagerGetDevices	0x1000067B8

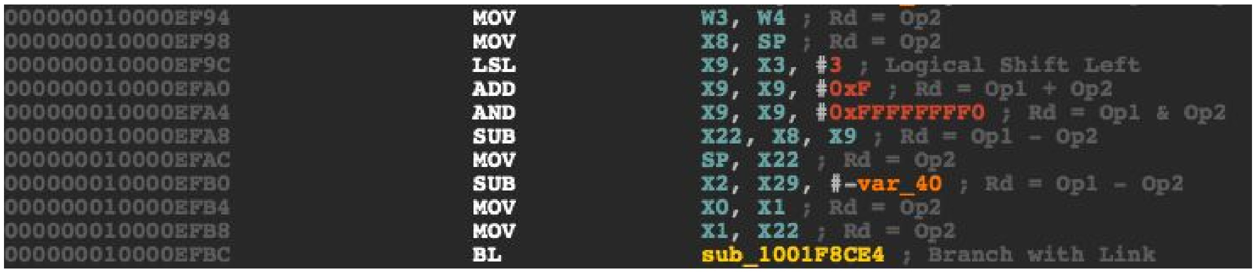
The handler does check msgh_size and msgh_bits.
Afterwards it calls inner function which does the rest of the logic.
The fifth argument of the function (offset 0x28 from the message header) is passed to the inner function as can be seen on figure 2



```
0000000100003254        LDR        W4, [X0,#0x28] ; Load from Memory
0000000100003258        MOV        X0, X8 ; Rd = Op2
000000010000325C        BL         sub_10000EF5C ; Branch with Link
```

Figure 2 - passing the given inputs into an inner function

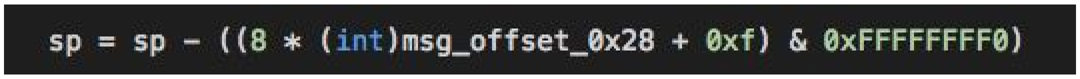
W4 - data from the message (offset 0x28) is 32bit size and reassigned to W3.



```
000000010000EF94        MOV        W3, W4 ; Rd = Op2
000000010000EF98        MOV        X8, SP ; Rd = Op2
000000010000EF9C        LSL        X9, X3, #3 ; Logical Shift Left
000000010000EFA0        ADD        X9, X9, #0xF ; Rd = Op1 + Op2
000000010000EFA4        AND        X9, X9, #0xFFFFFFFF0 ; Rd = Op1 & Op2
000000010000EFA8        SUB        X22, X8, X9 ; Rd = Op1 - Op2
000000010000EFAC        MOV        SP, X22 ; Rd = Op2
000000010000EFB0        SUB        X2, X29, #-var_40 ; Rd = Op1 - Op2
000000010000EFB4        MOV        X0, X1 ; Rd = Op2
000000010000EFB8        MOV        X1, X22 ; Rd = Op2
000000010000EFBC        BL         sub_1001F8CE4 ; Branch with Link
```

Figure 3 - further usage of input parameters

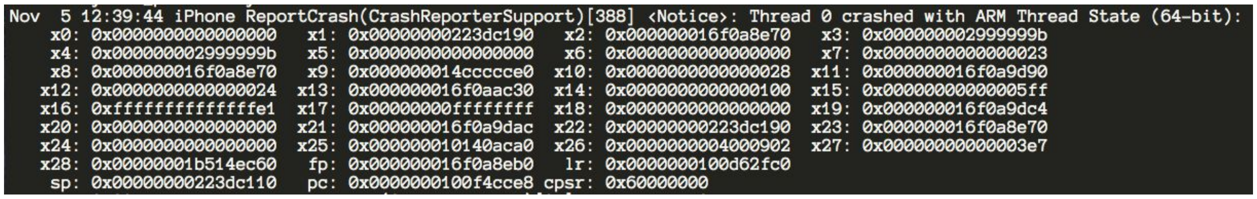
As you can see in the assembly above SP is recalculated by the user input without any bound checks or size. A pseudocode of this calculation is shown in figure 4.



```
sp = sp - ((8 * (int)msg_offset_0x28 + 0xf) & 0xFFFFFFFF0)
```

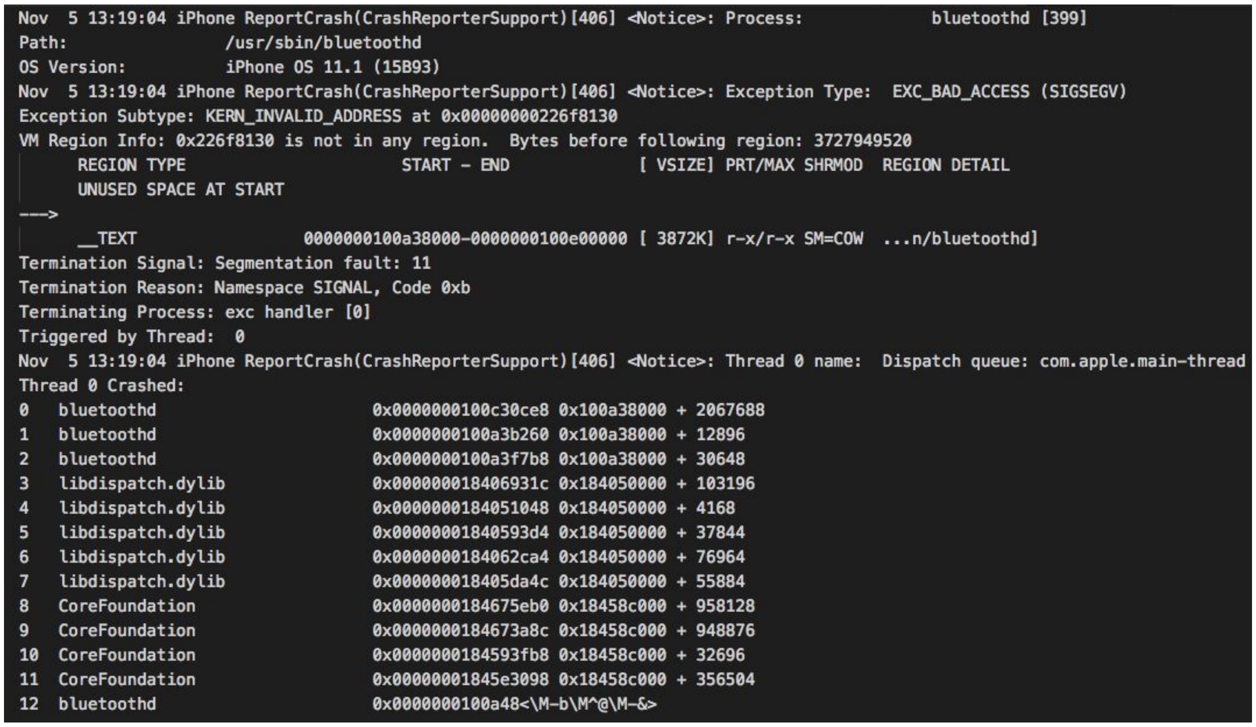
Figure 4 - SP manipulation from user input.

An attacker can take advantage of that vulnerability to hijack the code execution in bluetoothd. An example for a crash dump is shown in figure 4.



```
Nov  5 12:39:44 iPhone ReportCrash(CrashReporterSupport)[388] <Notice>: Thread 0 crashed with ARM Thread State (64-bit):
    x0: 0x0000000000000000   x1: 0x00000000223dc190   x2: 0x000000016f0a8e70   x3: 0x000000002999999b
    x4: 0x000000002999999b   x5: 0x0000000000000000   x6: 0x0000000000000000   x7: 0x0000000000000023
    x8: 0x000000016f0a8e70   x9: 0x000000014cccce0  x10: 0x0000000000000028  x11: 0x000000016f0a9d90
   x12: 0x0000000000000024  x13: 0x000000016f0aac30  x14: 0x0000000000000100  x15: 0x00000000000005ff
   x16: 0xffffffffffffffe1  x17: 0x00000000ffffffff  x18: 0x0000000000000000  x19: 0x000000016f0a9dc4
   x20: 0x0000000000000000  x21: 0x000000016f0a9dac  x22: 0x00000000223dc190  x23: 0x000000016f0a8e70
   x24: 0x0000000000000000  x25: 0x0000000010140aca0  x26: 0x0000000004000902  x27: 0x00000000000003e7
   x28: 0x00000001b514ec60   fp: 0x000000016f0a8eb0   lr: 0x0000000100d62fc0
    sp: 0x00000000223dc110   pc: 0x0000000100f4cce8 cpsr: 0x60000000
```

Figure 4 - Thread state for the crash

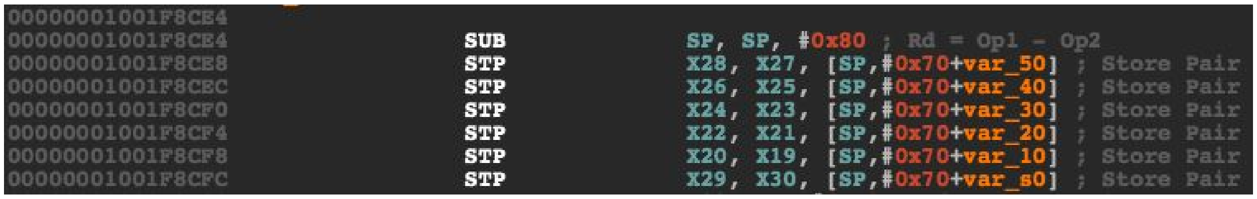


```
Nov  5 13:19:04 iPhone ReportCrash(CrashReporterSupport)[406] <Notice>: Process:           bluetoothd [399]
Path:               /usr/sbin/bluetoothd
OS Version:         iPhone OS 11.1 (15B93)
Nov  5 13:19:04 iPhone ReportCrash(CrashReporterSupport)[406] <Notice>: Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x00000000226f8130
VM Region Info: 0x226f8130 is not in any region.  Bytes before following region: 3727949520
      REGION TYPE                      START - END             [ VSIZE] PRT/MAX SHRMOD  REGION DETAIL
      UNUSED SPACE AT START
--->
      __TEXT                 0000000100a38000-0000000100e00000 [ 3872K] r-x/r-x SM=COW  ...n/bluetoothd]
Termination Signal: Segmentation fault: 11
Termination Reason: Namespace SIGNAL, Code 0xb
Terminating Process: exc handler [0]
Triggered by Thread:  0
Nov  5 13:19:04 iPhone ReportCrash(CrashReporterSupport)[406] <Notice>: Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   bluetoothd                    0x0000000100c30ce8 0x100a38000 + 2067688
1   bluetoothd                    0x0000000100a3b260 0x100a38000 + 12896
2   bluetoothd                    0x0000000100a3f7b8 0x100a38000 + 30648
3   libdispatch.dylib             0x000000018406931c 0x184050000 + 103196
4   libdispatch.dylib             0x0000000184051048 0x184050000 + 4168
5   libdispatch.dylib             0x00000001840593d4 0x184050000 + 37844
6   libdispatch.dylib             0x0000000184062ca4 0x184050000 + 76964
7   libdispatch.dylib             0x000000018405da4c 0x184050000 + 55884
8   CoreFoundation                0x0000000184675eb0 0x18458c000 + 958128
9   CoreFoundation                0x0000000184673a8c 0x18458c000 + 948876
10  CoreFoundation                0x00000001845593fb8 0x18458c000 + 32696
11  CoreFoundation                0x00000001845e3098 0x18458c000 + 356504
12  bluetoothd                    0x0000000100a48<\M-b\M^@\M-&>
```

Figure 5 - a crash dump example for bluetoothd

As you can see in the registers from the traceback above SP is now controlled by the user from sandboxed application.

The actual crash is when trying to store registers to memory controlled by the user (seen on figure 6), but when exploited appropriately, an attacker can cause type confusion of objects on the stack, hijack the execution by spraying the stack with an arbitrary return address and more.



```
00000001001F8CE4
00000001001F8CE4                 SUB             SP, SP, #0x80 ; Rd = Op1 - Op2
00000001001F8CE8                 STP             X28, X27, [SP,#0x70+var_50] ; Store Pair
00000001001F8CEC                 STP             X26, X25, [SP,#0x70+var_40] ; Store Pair
00000001001F8CF0                 STP             X24, X23, [SP,#0x70+var_30] ; Store Pair
00000001001F8CF4                 STP             X22, X21, [SP,#0x70+var_20] ; Store Pair
00000001001F8CF8                 STP             X20, X19, [SP,#0x70+var_10] ; Store Pair
00000001001F8CFC                 STP             X29, X30, [SP,#0x70+var_s0] ; Store Pair
```

Figure 6 - storing to the stack and causing a crash after an attacker modifies it

This means an attacker can control the stack pointer of bluetoothd, therefore allowing execution hijacking on bluetoothd.

POC is attached to the email - the code show the bug on message id 16 but will work the same with different ids as mentioned in the table above.

Unfortunately, mitigations such as stack cookie are irrelevant in this particular case because the stack cookie is being checked after all the logics and use of the stack, so it is still possible to leverage the vulnerability despite the stack cookie.

Solution

Should not do pointer calculations on stack pointers without having check of boundaries.

This bug is subject to a 90 day disclosure deadline, after which the bug report will be publicized.