CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE – BlueFrag

کسری عبداللهی سروی: ۹۷۱۰۶۱۲۱

محمد حیدری:۹۷۱۱۰۰۷۱

امروزه تقریبا تمام موبایل هایی که استفاده می کنیم از بلوتوث استفاده می کنند. این قابلیت توسط سخت افزار بلوتوث ای که توسط یک vendor ارائه می شود به یک گوشی هوشمند داده می شود. همچنین این سخت افزار برای کار کردن یک firm ware ای دارد که توسط همان vendor ارائه می شود.

آسیب پذیری ای که می خواهیم درباره اش صحبت کنیم مربوط به یک اشتباه در firm ware بلوتوث Broadcom در android 8 و android 9 است

برای درک بهتر این آسیب پذیری ابتدا کمی درباره شیوه ی کارکرد بلوتوث توضیح می دهیم:

بلوتوث یک ارتباط به شیوه ی master-slave ایجاد می کند. لازم است که یک دستگاهی که از بلوتوث پشتیبانی می کند بتواند با خود Bluetooth device ارتباط برقرار بکند. این ارتباط به وسیله ی لایه ی L2CAP (Logical Link Control and Adaptation Protocol) برقرار می شود. این لایه وظیفه دارد که داده هایی که از مثلا یک app می آیند (به طور کلی داده های host) را تقسیم بندی (fragmentation) بکند، داده هایی که به صورت بسته (packet) هایی جدا از هم آمدند را به هم متصل کند (reassembling) و ...

این لایه باید بتواند داده ها را بیت به بیت به Bluetooth device برساند. این کار وظیفه ی لایه ی HCI (Host Controller Interface) است. این لایه وظیفه ی دارد که دستور ها (commands) و رویداد ها (events) را بین host و controller آن Bluetooth device جا به جا کند.

آسیب پذیری ای که قرار است درباره اش توضیح بدهیم به ACL در بلوتوث نیز ارتباط دارد.

ACL(Asynchronous Connection Link) یک لینک انتقال داده از یک device به device دیگر است که به صورت یک پروتوکل پیاده سازی شده است.

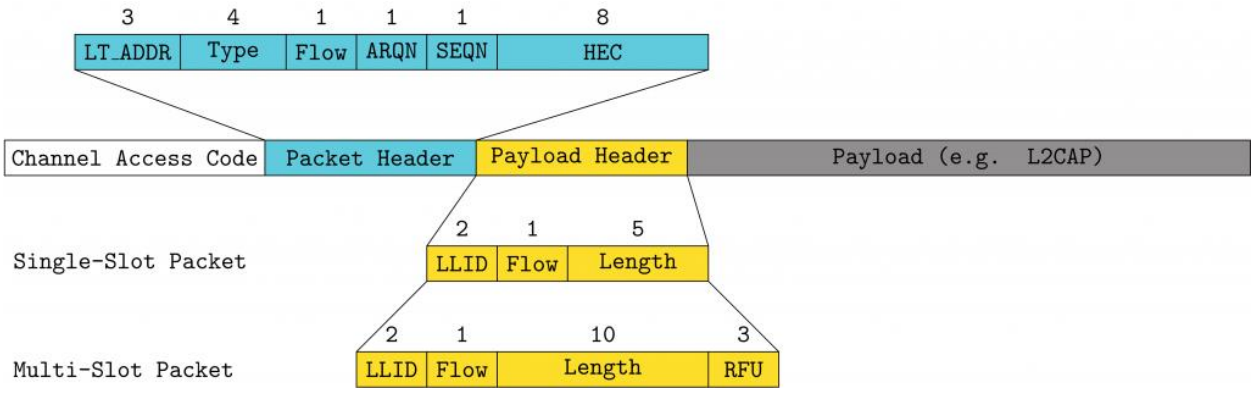
فرایند پیدا شدن bug

(توجه: توی فرایند توضیح این آسیب پدیری یه سری از جزئیاتی که دیگه خیلی کوچیک بودن رو نیاوردیم، به خاطر همین مقاله ی اصلی مربوط به این آسیب پذیری در اینجا یه خورده مفصل تره)

این آسیب پذیری با روش fuzzing کشف شد. Fuzzing یک روش است برای پیدا کردن آسیب پذیری ها (مخصوصا آسیب پذیری های مربوط به برنامه های نزدیک به سخت افزار و بالاخص آسیب پذیری های buffer overflow مانند انواع memory corruption و ...)

Fuzzing به این صورت است که داده ای که کاربر روی آن کنترل دارد به صورت تقریبا رندوم و چیز هایی که آن برنامه انتظار ندارد، به برنامه داده می شود و از روی رفتار برنامه می توان فهمید که آن برنامه آسیب پذیر هست یا خیر (برای مثال تصادفا ورودی ای به برنامه داده می شود که باعث می شود برنامه بخواهد به قسمتی غیر مجاز از حافظه دسترسی داشته باشد و سیستم عامل جلویش را می گیرد و ما یک ارور segmentation fault می گیریم و می فهمیم که احتمالا می توانیم یک آسیب پذیری memory corruption را exploit کنیم)

در کشف این آسیب پذیری ACL فاز شد به این صورت که packet header و payload header به صورت bit flip فاز می شوند. این کار با استفاده از hook کردن یک تابع در firmware اتفاق می افتد که این امکان را می دهد که بتوانیم کل packet را تغییر قبل از ارسال تغییر بدهیم.



به وسیله ی این fuzzer که بیت های header را در android device ما flip می کند و به وسیله ی ابزار l2ping (ابزاری که یک ارتباط ساده ی L2CAP ایجاد می کند و پیغام های echo را برای Bluetooth device ای که آدرسش را می دهیم ارسال می کند) در لینوکس این fuzzing انجام می شود. هنگام تلاش برای crash کردن firmware در اندروید Bluetooth deamon به ورودی ای خاص crash می کند و چنین پیغام خطایی در لاگ های اندروید نوشته می شود:

```
pid: 14808, tid: 14858, name: HwBinder:14808_  >>> com.android.bluetooth
<<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x79cde00000
    x0  00000079d18360e1  x1  00000079cddfffcb  x2  ffffffffffff385ef  x3
00000079d18fda60
    x4  00000079cdd3860a  x5  00000079d18360df  x6  0000000000000000  x7
0000000000000000
    x8  0000000000000000  x9  0000000000000000  x10 0000000000000000  x11
0000000000000000
    x12 0000000000000000  x13 0000000000000000  x14 ffffffffffffffff  x15
2610312e00000000
    x16 00000079bf1a02b8  x17 0000007a5891dcb0  x18 00000079bd818fda  x19
00000079cdd38600
    x20 00000079d1836000  x21 0000000000000097  x22 00000000000000db  x23
00000079bd81a588
    x24 00000079bd819c60  x25 00000079bd81a588  x26 0000000000000028  x27
0000000000000041
    x28 0000000000002019  x29 00000079bd819df0
    sp  00000079bd819c50  lr  00000079beef4124  pc  0000007a5891ddd4

backtrace:
    #00 pc 000000000001ddd4  /system/lib64/libc.so (memcpy+292)
```

```
    #01 pc 0000000000233120  /system/lib64/libbluetooth.so
(reassemble_and_dispatch(BT_HDR*) [clone .cfi]+1408)
    #02 pc 000000000022fc7c  /system/lib64/libbluetooth.so
(BluetoothHciCallbacks::aclDataReceived(android::hardware::hidl_vec<unsign
ed char> const&)+144)
    [...]
```

این خطا نشان می دهد که احتمالا یک دسترسی غیر مجاز به بخشی از حافظه اتفاق افتاده است. به یک پیاده سازی ساده از memcpy توجه کنید:

```
void *memcpy(char *dest; char *src, size_t *n) {
  for (size_t i=0; i<n; i++)
    dst[i] = src[i];
}
```

این تابع یک ورودی * size_t می گیرد که می گوید چقدر از ابتدای src را در dest بنویسد. دقت کنید که size_t یک unsigned است، بنابراین اگر یک مقدار منفی را در آن بریزیم به شکل یک عدد صحیح مثبت بسیار بزرگ تفسیر می شود و باعث می شود که تا جایی که ممکن است این copy انجام شود و به محض رسیدن به قسمتی که نباید در دسترس Bluetooth deamon باشد، اندروید این deamon را متوقف می کند و ما یک segmentation fault دریافت می کنیم.

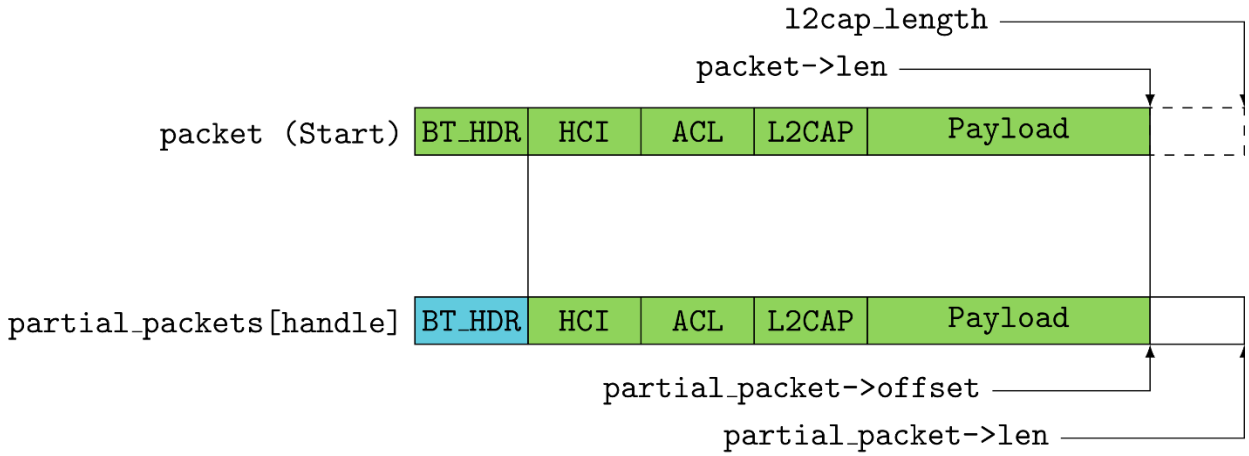
در این جا نیز همین اتفاق افتاده و یک مقدار منفی برای n ایجاد شده است.

Fragmentation در L2CAP

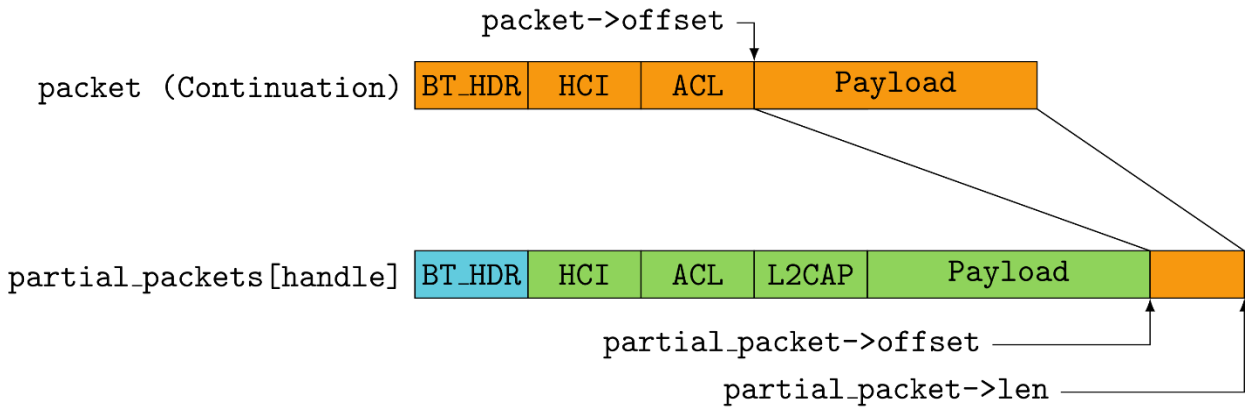
همانطور که در توضیحات ابتدایی ذکر شد packet هایی که از controller به host میروند (و برعکس) packet های L2CAP هستند. این packet ها نیز به وسیله ی packet های ACL از یک device به دیگری می رود.

Packet های ACL یک سایز بیشینه ای دارند که در firmware ای که Broadcom ارائه داده این سایز ۱۰۲۱ است. پس packet های L2CAP ای که سایزشان به گونه ای است که باعث شود packet های ACL سایزشان از ۱۰۲۱ بیشتر شود باید قطعه قطعه شودند (fragmentation). همچنین یک packet از نوع L2CAP که اندازه اش از ۱۰۲۱ بزرگ تر است باید reassemble شود چون می دانیم که قبلا قطعه قطعه شده است. این دو کار باعث می شود که L2CAP بسیار مستعد آسیب پذیری heap overflow باشد (هم در host و هم در controller)

Packet های L2CAP که قطعه قطعه آمده اند، این قطعه هایشان در یک map به نام partial_packets ذخیره می شوند که کلید آن یک conncetion handle است



در یک ACL packet نیز در header این یک بیتی وجود دارد که نشان می دهد این packet یک fragment از packet قبلی است (بیت ۱۲ ام header این packet)



همچنین انتهای آخرین fragment ای که رسیده و ذخیره شده نیز در یک متغیری (partial_packet -> offset) ذخیره می شود و وقتی که fragment بعدی رسید به وسیله ی این offset به انتهای fragment قبلی concat می شود

کد مربوط به این فرایند reassemble را در پایین می بینید:

```
static void reassemble_and_dispatch(UNUSED_ATTR BT_HDR *packet) {
      [...]
      packet->offset = HCI_ACL_PREAMBLE_SIZE;
      uint16_t projected_offset =
          partial_packet->offset + (packet->len - HCI_ACL_PREAMBLE_SIZE);
      if (projected_offset >
          partial_packet->len) {  // len stores the expected length
        LOG_WARN(LOG_TAG,
            "%s got packet which would exceed expected length of %d."
            "Truncating.",
            __func__, partial_packet->len);
```

```
        packet->len = partial_packet->len - partial_packet->offset;
        projected_offset = partial_packet->len;
      }
      memcpy(partial_packet->data + partial_packet->offset,
         packet->data + packet->offset, packet->len - packet->offset);

      [...]
}
```

بخشی که باعث منفی شدن size_t *n در memcpy می شود همین قطعه کد است.

در شرایط که یک packet ای دریافت شود که فقط ۲ بایت دیگر از آن باقی مانده است و packet بعدی طولش بیشتر از حد مورد انتظار باشد کد بالا مقدار packet -> len را برای جلوگیری از truncate buffer overflow می کند. در این فرایند مقدار partial_packet -> offset که در بالا برابر با HCI_ACL_PREAMBLE_SIZE که برابر با ۴ است قرار داده می شود و از مقدار partial_packet -> len که ۲ است کم می شود و مقدار ۲- را تولید می کند.

این bug به نظر می رسد که اصلا exploitable نباشد زیرا یک memcpy بی نهایت اتفاق می افتد که در آخر به یک crash می رسد، اما آسیب پذیری دیگری نیز پیدا شد

با بررسی بیشتر به crash ای (crash زیر) برخورد شد که به نظر نمی رسد که مربوط به یک حلقه ی بی پایان باشد.

```
pid: 14530, tid: 14579, name: btu message loo  >>> com.android.bluetooth
<<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x7a9e0072656761
    x0  0000007ab07d72c0  x1  0000007ab0795600  x2  0000007ab0795600  x3
0000000000000012
    x4  0000000000000000  x5  0000007a9e816178  x6  fefeff7a3ac305ff  x7
7f7f7f7f7fff7f7f
    x8  007a9e0072656761  x9  0000000000000000  x10 0000000000000020  x11
0000000000002000
    x12 0000007aa00fc350  x13 0000000000002000  x14 000000000000000d  x15
0000000000000000
    x16 0000007b396f6490  x17 0000007b3bc46120  x18 0000007a9e81542a  x19
0000007ab07d72c0
    x20 0000007ab0795600  x21 0000007a9e817588  x22 0000007a9e817588  x23
000000000000350f
    x24 0000000000000000  x25 0000007ab07d7058  x26 000000000000008b  x27
0000000000000000
    x28 0000007a9e817588  x29 0000007a9e816340
    sp  0000007a9e8161e0  lr  0000007a9fde0ca0  pc  0000007a9fe1a9a4

backtrace:
    #00 pc 00000000003229a4  /system/lib64/libbluetooth.so
(list_append(list_t*, void*) [clone .cfi]+52)
    #01 pc 00000000002e8c9c  /system/lib64/libbluetooth.so
(l2c_link_check_send_pkts(t_l2c_linkcb*, t_l2c_ccb*, BT_HDR*) [clone
.cfi]+100)
    #02 pc 00000000002ea25c  /system/lib64/libbluetooth.so
(l2c_rcv_acl_data(BT_HDR*) [clone .cfi]+1236)
    [...]
```

برای بررسی بیشتر چنین آزمایشی ترتیب داده شد:

۱. یک L2CAP packet با payload تماما ’A‘ می فرستیم که ۲ بایتش باقی مانده و باید در packet بعدی فرستاده شود
۲. در packet دوم با طولی بیشتر از ۲ که مورد انتظار است با payload تماما ’B‘ می فرستیم

این همان شرایطی است که باید memcpy با مقدار ۲- فراخوانی بشود و احتمالا یک حلقه ی بی پایان ایجاد کند ولی نتیجه ی ارسال این دو packet با gdb بررسی شد و چنین بود

Packet اول:

```
Thread 28 "HwBinder:14061_" hit Breakpoint 1, 0x0000007cce1afbc0 in reassemble_and_dispatch(BT_HDR*) [clone .cfi] ()
   from target:/system/lib64/libbluetooth.so
7cda585ba0: 0011 5200 0000 0000 0b20 4e00 4c00 0100  ..R..... N.L...
7cda585bb0: 0801 4800 4141 4141 4141 4141 4141 4141  ..H.AAAAAAAAAAA           BT_HDR
7cda585bc0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA          HCI Hdr.
7cda585bd0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA          ACL Hdr.
7cda585be0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA          L2CAP Hdr.
7cda585bf0: 4141 4141 4141 4141 4141 6572 3d31 2c00  AAAAAAAAAAer=1,.          Payload
7cda585c00: 0000 0000 6d00 2e00 3a00 0000 0000 0000  ....m...:.......
7cda585c10: 7500 6e00 6700 2e00 6100 6e00 6400 7200  u.n.g...a.n.d.r.

Thread 28 "HwBinder:14061_" hit Breakpoint 4, 0x0000007cce1afff0 in reassemble_and_dispatch(BT_HDR*) [clone .cfi] ()
   from target:/system/lib64/libbluetooth.so
$1 = 0x7cda587228
$2 = 0x7cda585ba8
$3 = 0x52
```

Packet دوم:

```
Thread 28 "HwBinder:14061_" hit Breakpoint 1, 0x0000007cce1afbc0 in reassemble_and_dispatch(BT_HDR*) [clone .cfi] ()
   from target:/system/lib64/libbluetooth.so
7cda585ba0: 0011 5200 0000 0000 0b10 4e00 4a00 0100  ..R.......N.J...
7cda585bb0: 0802 4600 4242 4242 4242 4242 4242 4242  ..F.BBBBBBBBBBBB          BT_HDR
7cda585bc0: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB          HCI Hdr.
7cda585bd0: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB          ACL Hdr.
7cda585be0: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB          L2CAP Hdr.
7cda585bf0: 4242 4242 4242 4242 4242 6572 3d31 2c00  BBBBBBBBBBer=1,.          Payload
7cda585c00: 0000 0000 6d00 2e00 3a00 0000 0000 0000  ....m...:.......
7cda585c10: 7500 6e00 6700 2e00 6100 6e00 6400 7200  u.n.g...a.n.d.r.

Thread 28 "HwBinder:14061_" hit Breakpoint 3, 0x0000007cce1b0120 in reassemble_and_dispatch(BT_HDR*) [clone .cfi] ()
   from target:/system/lib64/libbluetooth.so
$4 = 0x7cda58727a
$5 = 0x7cda585bac
$6 = 0xfffffffffffffffe
```

ولی عکس پایین نشان می دهد که ظاهرا چنین حالتی ۶۶ بایت آخر از partial packet را خراب می کند و برنامه هم به اجرایش ادامه می دهد:

```
Thread 28 "HwBinder:14061_" hit Breakpoint 2, 0x0000007cce1ac460 in dispatch_reassembled(BT_HDR*) [clone .cfi] ()
   from target:/system/lib64/libbluetooth.so
7cda587220: 0011 5400 0000 0000 0b20 5000 4c00 0100  ..T..... P.L...
7cda587230: 0801 4800 4141 4141 6c00 7500 6500 7400  ..H.AAAAl.u.e.t.          BT_HDR
7cda587240: 6f00 6f00 7400 6800 2e00 4900 4200 6c00  o.o.t.h...I.B.l.          HCI Hdr.
7cda587250: 7500 6500 7400 6f00 6f00 7400 6800 0000  u.e.t.o.o.t.h...          ACL Hdr.
7cda587260: 6400 0000 0000 4141 4a00 0100 0809 0011  d.....AAJ.......          L2CAP Hdr.
7cda587270: 0200 0400 0000 0b10 4141 4a00 0100 0802  ........AAJ.....          Payload
7cda587280: 4600 4242 4242 4242 4242 4242 4242 4242  F.BBBBBBBBBBBBBB
7cda587290: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
```

علت چنین رفتاری مربوط به رفتار عجیب پیاده سازی memcpy در libc در اندروید است. پیاده سازی memcpy به آن سادگی که در بالا کدی برایش آوردیم نیست و برای بهینه انجام دادن عمل کپی پیچیدگی هایی دارد که باعث می شود می

شود رفتار عجیبی برای طول ها به مقدار منفی داشته باشد که منجر می شود که بتوانیم ۶۶ بایت آخر از l2ping request امان را به وسیله ی آن چه که قبل از packet دوم است overwrite کنیم

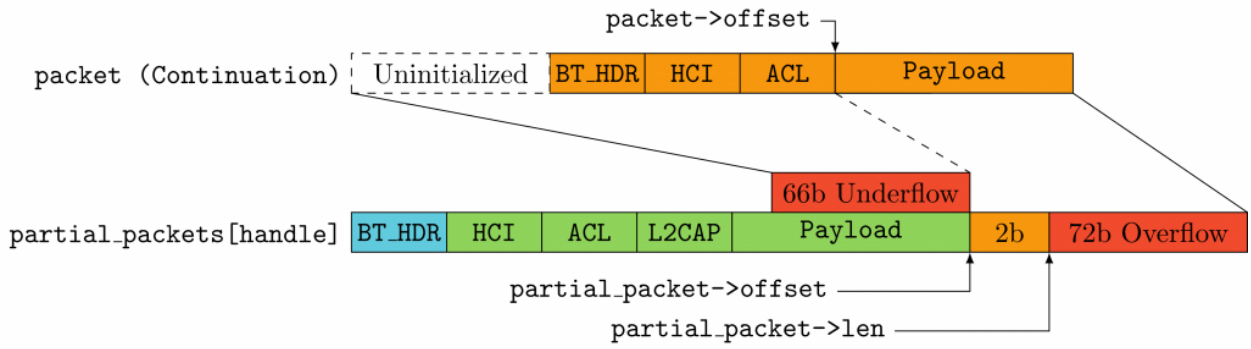
کد زیر و نتیجه ی آن نمایش دهنده ی proof of concept از این رفتار memcpy است (این نتیجه مربوط به emulation ای از معماری aarch64 است که به وسیله ی Unicorn انجام شده است:

```
int main(int argc, char **argv) {
    if (argc < 3) {
        printf("usage %s offset_dst offset_src\n", argv[0]);
        exit(1);
    }

    char *src = malloc(256);
    char *dst = malloc(256);

    printf("src=%p\n", src);
    printf("dst=%p\n", dst);

    for (int i=0; i<256; i++) src[i] = i;
    memset(dst, 0x23, 256);

    memcpy( dst + 128 + atoi(argv[1]),
            src + 128 + atoi(argv[2]),
            0xfffffffffffffffe );

    //Hexdump
    for(int i=0; i<256; i+=32) {
        printf("%04x:  ", i);
        for (int j=0; j<32; j++) {
            printf("%02x", dst[i+j] & 0xff);
            if (j%4 == 3) printf(" ");
        }
        printf("\n");
    }
}
```

نتیجه:

```
beyond0:/data/local/tmp $ ./a.out 0 0
src=0x713f42d000
dst=0x713f42d100
Segmentation fault
139|beyond0:/data/local/tmp $ ./a.out 4 4
src=0x7012e2d000
dst=0x7012e2d100
0000:  23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323
0040:  23234243 44454647 48494a4b 4c4d4e4f 50515253 54555657 58595a5b 5c5d5e5f 60616263 64656667 68696a6b 6c6d6e6f 70717273 74757677 78797a7b 7c7d7e7f
0080:  80812323 84858687 88898a8b 8c8d8e8f 90919293 94959697 98999a9b 9c9d9e9f a0a1a2a3 a4a5a6a7 a8a9aaab acadaeaf b0b1b2b3 b4b5b6b7 b8b9babb bcbdbebf
00c0:  c0c1c2c3 c4c5c6c7 c8c9cacb cccdcecf 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323 23232323
```

شکل زیر فرایند overwrite کردن به وسیله ی packet دوم را بهتر نمایش می دهد:



طبق شکل بالا ما می توانیم ۶۴ بایت آخر packet را با استفاده از ۲۰ بایت header ها (BT_HDR, HCI, ACL) و بخش uninitialized از memory overwrite کنیم. محتوای بخش uninitialized وابسته به مکان buffer مربوط به packet دوم و نتیجتا سایز آن است. با چند بار l2ping request می توانیم این محتوا را کنترل کنیم. همچنین با کوتاه کردن طول packet می توانیم کل struct مربوط به packet را اعم از header هایش را کنترل کنیم

علاوه بر این می توانیم با تغییر طول packet مقدار بیشتری memory leak داشته باشیم. با استفاده از این leak می توانیم به نوعی ASLR را دور بزنیم، به این صورت که در یک library به اسم libicuuc.so یک struct وجود دارد که در آن چند pointer به چند تابع وجود دارد که با استفاده از آن ها می توانیم به محاسبه ی offset ها ASLR را دور بزنیم

(نکته برای کسانی که نمی دانند ASLR چیست: ASLR مخفف Address Space Layout Randomization است و یک شیوه ی محافظت از حملات buffer overflow است که باعث می شود adversary نتواند بفهمد که یک قطعه کد در کدام آدرس قرار گرفته پس اگر هم بتواند pc را کنترل کند نخواهد توانست به تکه کدی که مورد نظرش است jump کند، حال در شرایطی که یک pointer به یک تکه از کد لو برود، مثلا با یک memory leak، ASLR دور زده شده است)

دقت کنید که تا این جا فقط راجع به لو رفتن بخشی از memory صحبت کردیم و به مرحله ی code execution نرسیدیم

در فرایند leak کردن memory علاوه بر این که در این مورد می توانیم مقادری قبل از buffer را underflow کنیم، می توانیم مقادیر بعد از buffer را نیز overflow کنیم. در این جا بعد از buffer می توانیم یک pointer که به وسیله ی رجیستر $X0 کنترل می شود را تغییر دهیم. به قطعه کد زیر توجه کنید:

```
dbc5c: f9400008 ldr x8, [x0] // We control X0
dbc60: f9400108 ldr x8, [x8]
dbc64: aa1403e1 mov x1, x20
dbc68: d63f0100 blr x8 // Brach to **X0
```

پس تا این جا هم ASLR را دور زدیم هم توانستیم جایی را پیدا کنیم که به وسیله ی کنترل آن به آدرسی که می خواهیم jump کنیم.

حال چیزی که باقی می ماند تولید کردن یک ROP chain برای اجرای یک shell code است.

(نکته برای کسانی که با ROP آشنایی ندارند: ROP یک تکنیک برای رسیدن به shell code زمانی است که NX در سیستم فعال است. ROP مخفف Return Oriented Programming است. برای فهم بیشتر ابتدا باید بفهمیم که NX چیست. NX یک روش برای جلوگیری از اجرای کد دلخواهی که کاربر به عنوان ورودی به برنامه داده است، یعنی با NX قسمتی که کاربر روی آن کنترل دارد Non Executable می شود و فقط همان بخش هایی که در برنامه بوده اند مانند library ها و خود کد اصلی برنامه Executable هستند. با ROP ما NX را دور می زنیم به این صورت که با استفاده از تکه های مختلف از همان کد هایی که executable هستند به وسیله ی چسباندنشان به یک دیگه به شیوه ای مناسب به shell code می رسیم. Chain در ROP chain تعداد return address اند که باعث می شود که مشخص کنیم ابتدا به کجا jump کنیم، بعد به کجا، بعد به کجا و ... تا این که در نهایت با مجموعه ی اجرای این تکه کد ها یک shell code مورد نظر اجرا بشود)

اکنون به ROP هم با توجه به ماکسیمم طول مجاز packet که به اندازه ی کافی بزرگ هست (برای ایجاد chain مناسب ما) رسیدیم. حال باید آدرس یک تابع مناسب برای اجرای shell code را پیدا کنیم (مانند تابع system یا تابع execve).

متاسفانه این دو تابع در libicuuc.so موجود نیستند و در library های دیگری هستند و فاصله ی library ها نیز از هم به وسیله ی ASLR رندوم است و ما نمی توانیم بفهمیم که library مربوط به هر کدام از این دو تابع کجاست که خود تابع در آن library را آدرسش را بدانیم. اما خوشبختانه در library های import شده یک تابع با نام dlsym وجود دارد. این تابع یک handle ورودی می گیرد (که مهم نیست) و اسم یک تابع را می گیرد و آدرسش را به ما بر می گرداند. به وسیله ی این تابع می توانیم آدرس system را پیدا کنیم و با استفاده از آن shell را بگیریم.

این بود نتیجه ی مجموع یک آسیب پذیری خیلی کوچک در کد تولید شده توسط Broadcom و یک آسیب پذیری کوچک دیگر در پیاده سازی memcpy در کتاب خانه ی اندروید که منجر شد به اجرای هر کد دلخواه با دسترسی Bluetooth deamon در اندروید :)))