Exploiting Heap Corruption due to Integer Overflow in Android libcutils

By Guang Gong
@oldfresher

Before reading this article, you'd better understand binder mechanism, bufferqueue of graphic system, je_malloc in Android. This article detail how to exploit CVE-2015-1528 to get system_server permission in Android.

1.where was the vulnerable code

The vulnerable code is at the function which is used to create native handle[1], every GraphicBuffer object contain a native handle point by which GraphicBuffer can share graphic memory across process.

```
28native_handle_t* native_handle_create(int numFds, int numInts)
29{
30    native_handle_t* h = malloc(
31            sizeof(native_handle_t) + sizeof(int)*(numFds+numInts));---------->Integer Overflow
32
33    h->version = sizeof(native_handle_t);
34    h->numFds = numFds;
35    h->numInts = numInts;
36    return h;
37}
```

When native_handle_create is called with the carefully constructed numFds and numInts such as 0xffffffff and 2, the allocated size "sizeof(native_handle_t) + sizeof(int)*(numFds+numInts)" will overflow, the subsequent writing to the allocated buffer will cause heap corruption. there are two functions that can cause corruption. one affect all the versions of android and the other affect only Android Lollipop.

Function affect all Android versions is as follows[2]

```
303status_t GraphicBuffer::unflatten(
304        void const*& buffer, size_t& size, int const*& fds, size_t& count) {
...
330        native_handle* h = native_handle_create(numFds, numInts);
331        memcpy(h->data,         fds,     numFds*sizeof(int));    ---------->Heap Corruption
332        memcpy(h->data + numFds, &buf[10], numInts*sizeof(int));
...
}
```

As the allocated size is smaller than expected. memcpy will lead to heap corruption.

Functions affect only Android Lollipop is as follows[3]this function is defined in all versions of Android, but it seems to be used only in Lollipop.

```
1210 native_handle* Parcel::readNativeHandle() const
...
1219    native_handle* h = native_handle_create(numFds, numInts);
1220    for (int i=0 ; err==NO_ERROR && i<numFds ; i++) {
1221        h->data[i] = dup(readFileDescriptor());
1222        if (h->data[i] < 0) err = BAD_VALUE;
1223    }
1224    err = read(h->data + numFds, sizeof(int)*numInts);      ---------------->Heap Corruption
...
1230    return h;
1231}
```

Similar as the previous function, Since the allocated size is smaller than expected. read will lead to heap corruption. I will use this function to demonstrate the exploiting process.
To make things worse, numFds,numInts and the buffer to be copied can be manipulated across process, which means a process with low privilege can write the heap of a process with high privileges using the controlled content, so this is exploitable.

2.the attack vector

If a process can get a binder proxy object with IGraphicProducer interface, this process can exploit the process which hold the real binder object by binder call. The key function to exploit successfully is setSidebandStream[4] of IGraphicProducer interface.

```
403       case SET_SIDEBAND_STREAM: {
404           CHECK_INTERFACE(IGraphicBufferProducer, data, reply);
405           sp<NativeHandle> stream;
406           if (data.readInt32()) {
407               stream = NativeHandle::create(data.readNativeHandle(), true);
408           }
409           status_t result = setSidebandStream(stream);
410           reply->writeInt32(result);
411           return NO_ERROR;
412       } break;
```

The code above is how BnGraphicBufferProducer handle SET_SIDEBAND_STREAM sent by other process, we can see readNativeHandle is called and numFds,numInts and the buffer to be copied can be passed to the attacked process.
the following is a figure of the attack scene.

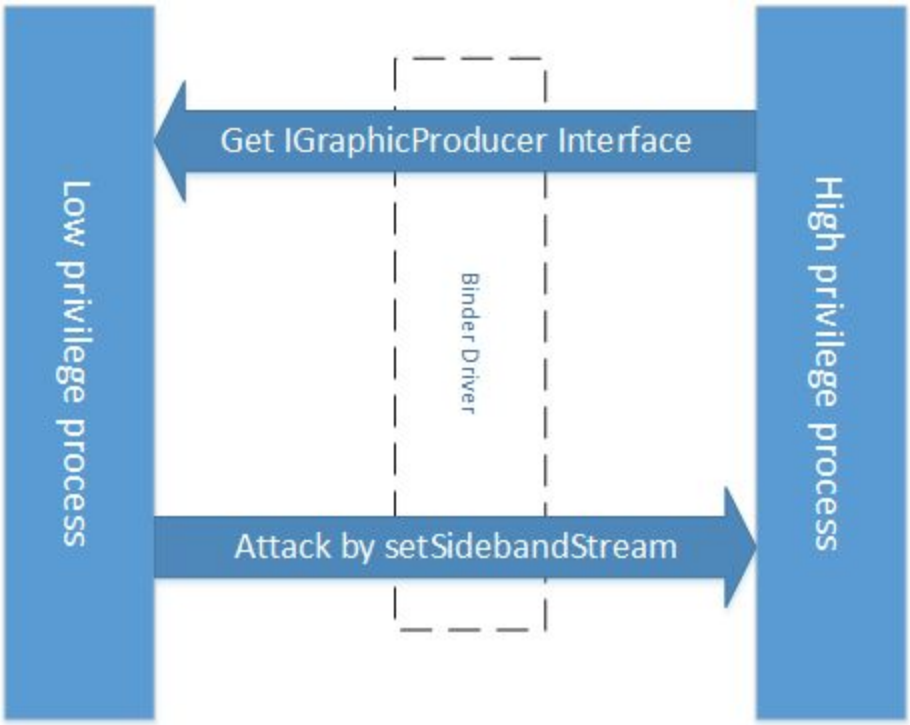

Figure1.attack scene

3.get system_server permission step by step

After validation, through an normal installed Apps, we can inject arbitrary code into mediaserver, surfaceflinger and system_server in order by this single vulnerability. please note that we have to take down these three process in order, for example, we get the proper permission to access surfaceflinger if and only if we conquer mediaserver.

```
ggong@ggong-pc:~/Downloads/pushsdk$ adb shell ps -Z | grep -E "system_server|su
u:r:surfaceflinger:s0          system    171   1     /system/bin/surfaceflinger
u:r:system_server:s0           system    696   197   system_server
u:r:mediaserver:s0             media     24465 1     /system/bin/mediaserver
```

Figure2. SELinux domains

Although surfaceflinger run as system user, because the exist of SELinux, surfaceflinger run in domain u:r:surfaceflinger:s0, this domain has little privilege excepting access graphic device, that is why we want to inject into system_server. system_server is the "kernel" of Android, Although we can't get root permission, we are "half-God" with the privilege of u:r:system_server:s0 domain. the following figure show by which binder calls we can get system_server permission finally.

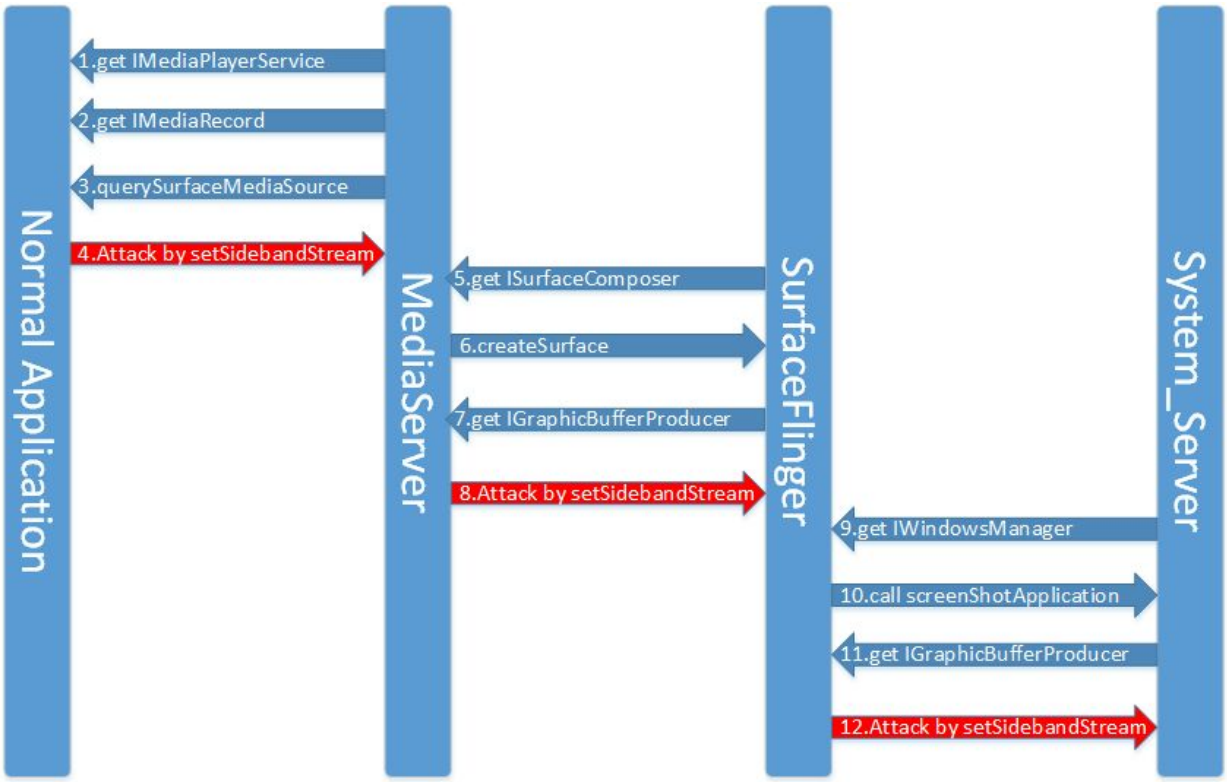

Figure 3. Get system_server permission by three steps

As the above figure shows, a normal Application can get an IGraphicProducer interface exposed by mediaserver through querySurfaceMediaSource, so normal Apps can take down mediaserver by setSidebandStream. As process meidaserver have ACCESS_SURFACE_FLINGER permission, so mediaserver can call createSurface to get An IGrphicProducer interface exposed by surfaceflinger, so injected code in mediaserver can take down surfaceflinger. surfaceflinger calling IWindowsManager.screenShotApplication will trigger system_server call captureScreen of ISurfaceComposer, which will call back to surfaceflinger with an argument IGraphicProducer expose by system_server, so the injected code in surfaceflinger can take down systemserver using this interface.

4.the detail of exploiting mediaserver

We have to take three steps to get system_server permission and each step is difficult as the exist of NX, ASLR, SELinux and multiple binder server threads combined with je_malloc features. I'll just show how to exploit media_server. Simply put, there are 5 steps to exploit mediaserver successfully.

1)control binder server threads

As we know, every thread is associated with a specific arena when using je_malloc[5], heap memory required in different threads will be allocated using different arenas, so, the small

memory allocated by different threads will be put in different chunks(In Android, the size of chunk is 1MB). Figure 4 shows the discreteness of the heap in je_malloc.

```
ggong@ggong-pc:~/.../mediaserver$ adbgetmaps mediaserver | grep libc_malloc
af800000-b2500000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b2700000-b3e00000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b4100000-b4300000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b4600000-b4700000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b4800000-b4a00000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b4d00000-b4e00000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b5000000-b5200000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b5300000-b5600000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b5800000-b5900000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b5b00000-b5c00000 rw-p 00000000 00:00 0          [anon:libc_malloc]
b6000000-b6200000 rw-p 00000000 00:00 0          [anon:libc_malloc]
```

Figure 4.heap distribution in je_malloc

binder server threads serve the binder call from binder clients, the number of binder server threads increase with the synchronous calling from the binder clients. the count of binder server threads in media server start with 4 and can be increased to 17 eventually.

```
ggong@ggong-pc:~/.../mediaserver$ adbgetstack medias | egrep "Binder|\"media"
"mediaserver" sysTid=2110
"Binder_1" sysTid=2138
"Binder_2" sysTid=2139
"Binder_3" sysTid=2140
"Binder_4" sysTid=2141
"Binder_5" sysTid=2324
"Binder_6" sysTid=2325
"Binder_7" sysTid=2326
"Binder_8" sysTid=2327
"Binder_9" sysTid=2328
"Binder_A" sysTid=2329
"Binder_B" sysTid=2330
"Binder_C" sysTid=2331
"Binder_D" sysTid=2332
"Binder_E" sysTid=2333
"Binder_F" sysTid=2334
"Binder_10" sysTid=2335
```

Figure 5.binder server threads in mediaserver

when a binder call is sent to mediaserver, system choose a binder server thread randomly to handle this call, in addition, different threads allocate memory in different thunk, so it's impossible to manipulate heap(heap FENGSHUI) by binder call if so many binder server thread are active. the solution is block all the binder server threads except one, so the definite thread will be chosen to handle the binder call and we can do heap fengshui. IGraphicProducer can be used to attach GraphicBuffer to bufferqueue. every bufferqueue can only store a specific amount of GraphicBuffer. when the count of attached buffer is larger than the specific amount,

the binder server thread will enter wait state and not handle other binder call,so we can use attachBuffer in IGraphicProducer to put binder server thread in wait thread.

2) leak heap content

Because the exist of ASLR, we need to leak address information from mediaserver. In je_malloc, same size memory allocation occupy the adjacent region, not like dl_malloc, there are no metadata between regions. we can allocate many normal native_handle in mediaserver by attachBuffer. The structure of native_handle_t is as follows, the allocated size is decided by numFds and numInts(in nexus 5, numFds is 2 and numInts is 12, and the allocated size is 80)

```
(gdb) pt native_handle_t
type = struct native_handle {
    int version;
    int numFds;
    int numInts;
    int data[4294967296];
}
```

numFds and numInts can be modified to a bigger value by setSidebandStream, when requestBuffer is called to get buffer associated with the modified native_handle, more heap memory will be leaked to the attack process.

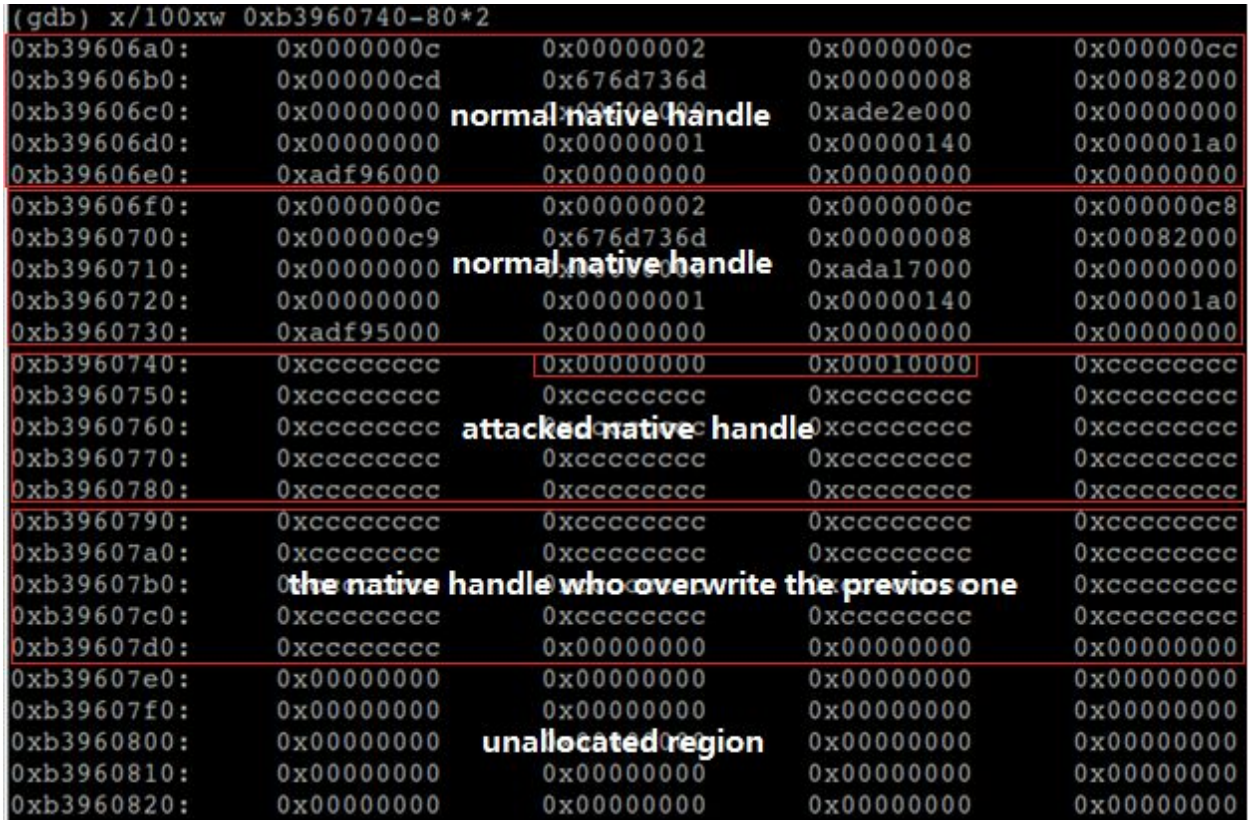

Figure 6.leak heap memory from mediaserver

As the above Figure shows, numFds and numInts of the attacked native handle are modified to 0 and 0x10000. when the GraphicBuffer associated with this handle is request by the attacker, 4*0x10000 bytes from 0xb396074c will be leaked from media server.

3)leak stack base address

Because the exist of NX, we need to use ROP to bypass it. ROP need to control the content of stack. If the stack base address can be leaked, we can use the vulnerability mentioned above to rewrite stack, so we can convert the heap corruption vulnerability to stack overwrite.

We already know how to leak memory from heap, so we can search heap to get the stack base address. The key structure to accomplish it is pthread_internal_t.

```
0xb652ec8c:   0xb424a080  0xb3b7c080  0x00000b58  0x00000a53
0xb652ec9c:   0xae8dcdb0  0x00000001  0xae7df000  0x000fe000
0xb652ecac:   0x00001000  0x00000000  0x00000000  0x00000000
0xb652ecbc:   0xb6e4700d  0xb3d48960  0x00000000  0xae7dd000
0xb652eccc:   0x00000001  0x00000000  0x00000000  0x00000000
0xb652ecdc:   0x00000000  0x00000000  0x00000000  0x00000000
```

the above address range shows the content of an pthread_internal_t, from which we know tid is 0xb58, pid is 0xa53, thread stack base is at 0xae7df000. So, we can get stack base address by search the leaked heap memory. the magic to search are stack_size(0x000fe000) and guard_size(0x00001000). As we know, generally new created thread has bigger thread id. New binder server thread are created when there is no thread available to handle the binder call. So, we can find several pthread_internal_t and choose the one with the biggest thread id, we have a maximum probability of finding the binder server thread which is in wait state. The stack backtrace of the waiting binder threads are known, it's as Figure 7, we can rewrite the return address to trigger the execution of ROP.

Figure 7.stack backtrace of the blocked thread

4)leak base address of shared library

We need the base address of shared libraries to execute ROP. it's easy to leak the base address of libui.so. Then only thing we need to do is find a GraphicBuffer from the leaked heap. There is a sub structure named android_native_base_t inside GraphicBuffer. we can caculate the base of libui.so from the function point incRef and decRef.

```
(gdb) pt/m android_native_base_t
type = struct android_native_base_t {
    int magic;
    int version;
    void *reserved[4];
    void (*incRef)(android_native_base_t *);
    void (*decRef)(android_native_base_t *);
}
```

it's better to get base address of libc.so to write shellcode. the GOT of libui.so contain the memcpy address, from which we can calculate the base address of libc.so. But it's a little difficult to read the content of the GOT because we can only leak specific size of contiguous memory from heap by modifying numFds and numInts, there are unmapped memory between heap address and the libui.so. I find another method to leak the content of libui.so. GraphicBuffer contain the point of native handle, if this point is modified to point to a faked native handle, we can leak the memory following the faked native handle. the size of the leaked memory is up to the numFds and numInts of the faked handle. we can get dlopen and dlsym address by reading the GOT of libc.so and use them in shellcode.

5)control the next allocation position of je_malloc

By construct specific numFds and numInts, we can only write the memory near native handle, as the stack is not adjacent to heap generally. We need to find a way to write to stack to trigger ROP.Luckily we can convert this vulnerability to write arbitrary value to arbitrary address by modifying the point table of tcache thread cache in je_malloc. each thread maintains a cache of small objects, this cache is stored in a structure named tcache_t. there are a point tables for every size class. When allocate small objects, je_malloc first try to find a cached memory in a specific point table. there are 31 size classes for small objects in the realization of je_malloc in Android, refer to the following gdb output.

```
(gdb) p je_small_bin2size_tab
$24 = {8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384,
448, 512, 640, 768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584}
```

the point tables of tcache is allocated in heap mixed with other normal allocated memory, so we can modify them. The following gdb output show one point table of tcache. When objects larger than 112 bytes and smaller than or equal 128 are allocated, this point table will be use. ncached-1 is the index of next allocated point of specific size. We can modify this value to point to stack, then when we allocated memory of specific size, the stack address will be returned, and writing to the allocated buffer will overwrite stack. So, we can execute ROP in the stack.

```
(gdb) p je_arenas[0].tcache_ql.qlh_first.tbins[11]
$9 = {tstats = {nrequests = 17}, low_water = 62, lg_fill_div = 1, ncached = 63, avail
= 0xb6003f60}
(gdb) x/63xw je_arenas[0].tcache_ql.qlh_first.tbins[11].avail
0xb6003f60:   0xb6057f80    0xb6057f00    0xb6057e80    0xb6057e00
0xb6003f70:   0xb6057d80    0xb6057d00    0xb6057c80    0xb6057c00
0xb6003f80:   0xb6057b80    0xb6057b00    0xb6057a80    0xb6057a00
0xb6003f90:   0xb6057980    0xb6057900    0xb6057880    0xb6057800
0xb6003fa0:   0xb6057780    0xb6057700    0xb6057680    0xb6057600
```

5.bypass SELinux to load so

Because the exist of SELinux, The file created by the normal application with label of app_data_file or apk_data_file. media server have no sepolicy to load binary with these label. So system and dlopen can't be use to execute binary in mediaserver. luckily, media has execmem permission

```
allow mediaserver self:process execmem;              ------------>SELinux policy
```

we can use mprotect to modify anonymous memory to executable so we can execute shellcode. I implement a mechanism of loading so from memory in shellcode in my PoC(that is complicated and i won't detail it in this article), So shared library with label app_data_file or apk_data_file can be passed to mediaserve as binary stream by the vulnerability, and then be loaded from memory.

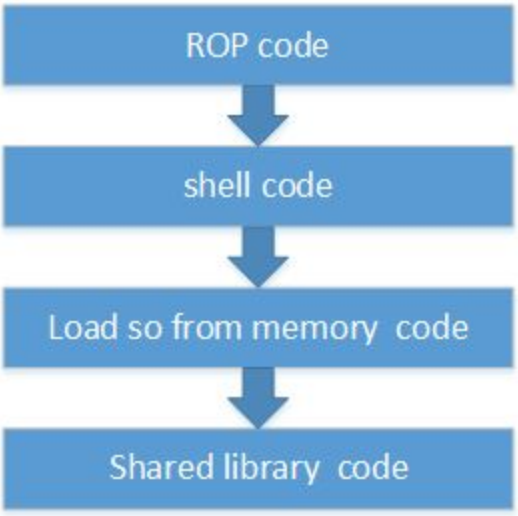

Figure 8. load shared library in mediaserver

As media server has no permission to execute /system/bin/sh, I also inject busybox to media server so we can execute simple command. If exploit successfully, we can get a shell from media server.

```
exploit successfully, enter shell
buffer len is 2302032, writed len is 2302032
success
input:id
uid=1013 gid=1005 groups=1006,1026,1031,3001,3002,3003,3007
input:whoami
whoami: unknown uid 1013
input:cat /proc/self/attrib/current
cat: can't open '/proc/self/attrib/current': No such file or directory
input:cat /proc/self/attr/current
u:r:mediaserver:s0input:cat /proc/self/attr/current
u:r:mediaserver:s0input:ls -l /data/misc/audio
total 8
-rw-------    1 1013     1005           154 Jan 30 09:05 mbhc.bin
-rw-------    1 1013     1005           536 Jan 30 09:05 wcd9320_anc.bin
input:
```

Figure 8. Shell with mediaserver permission

6. exploit surfaceflinger and system_server

Exploiting surfaceflinger and system_server is similar as exploiting media_server except a few exception.
1.surfaceflinger don't have execmem permission, so we can only use ROP in surfaceflinger.
2.the module base addresses are the same between system_server and normal application as they both are forked by Zygote process, it's not necessary to leak them.
I've implement the whole exploit process in my PoC. the link of the PoC is at https://github.com/secmob/PoCForCVE-2015-1528

[References]

[1]http://androidxref.com/5.0.0_r2/xref/system/core/libcutils/native_handle.c#29
[2]http://androidxref.com/5.0.0_r2/xref/frameworks/native/libs/ui/GraphicBuffer.cpp#303
[3]http://androidxref.com/5.0.0_r2/xref/frameworks/native/libs/binder/Parcel.cpp#1210
[4]http://androidxref.com/5.0.0_r2/xref/frameworks/native/libs/gui/IGraphicBufferProducer.cpp#403
[5]http://www.phrack.org/issues/68/10.html